UTEC-82-103

(13)

Second Semiannual Technical Report

TRANSFORMATION of ADA PROGRAMS INTO SILICON

82 Mar 1 – 82 Oct 31

Elliott I. Organick, Principal Investigator
(801) 581-6087

Contractor: The University of Utah

Date of Contract: 81 SEPT 1

Expiring: 83 AUG 31

Sponsored by

Defense Advanced Research Projects Agency (DoD)
ARPA Order No. 4305

Under Contract No. MDA 903-81-C-0411, issued by
Defense Supply Service – Washington, Washington DC 20310

November 1982

Table of Contents

SECURITY CLASSIFICATION OF THIS PAGE (*When Date Entered*)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER UTEC-82-[illegible] 103	2. GOVT ACCESSION NO. AD A122 967	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (*and Subtitle*) TRANSFORMATION OF ADA PROGRAMS INTO SILICON		5. TYPE OF REPORT & PERIOD COVERED 2nd semi-annual ~~81 Sept 1 - 82 Feb 28~~ 1 Mar - 31 Oct 82
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(*s*) Dr. E. Organick, Dr. G. Lindstrom, D. K. Smith, Dr. Subrahmanyam, T. Carter		8. CONTRACT OR GRANT NUMBER(*s*) MDA 903-81-C-0411
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Utah Computer Science Department Salt Lake City Utah 84112		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 1001/1122
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency (DoD) 1400 Wilson Boulevard Washington, D.C. 22209		12. REPORT DATE Nov ~~March~~ 1982
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(*if different from Controlling Office*) Defense Supply--Service Washington Rm 1d-245, The Pentagon Washington, D.C. 20310		15. SECURITY CLASS. (*of this report*) unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (*of this Report*)

This document has been approved for public release and sale; its distribution is unlimited.

17. DISTRIBUTION STATEMENT (*of the abstract entered in Block 20, if different from Report*)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (*Continue on reverse side if necessary and identify by block number*) control unit, CADDET, SPICE Internet protocol, submodules, Ada-to-silicon, transformation methelodogies, high level program specifications, DoD Internet Protocol, special function architecture, ADA packages & tasks, VLSI synthesis, program formal specifications, device modeling, switched capacitor filter, stored logic array, logic simulator, hand shake, speed-independent, one-hot, portable standard LISP, silicon compiler, VLSI

This report outlines the beginning steps taken in an integrated research effort toward the development of a methodology, and supporting systems, for transforming Ada programs, or program units, (directly) into corresponding VLSI systems. The time seems right to expect good results. The need is evident; special purpose systems should be realistic alternatives where simplicity, speed, reliability, and security are dominant factors. Success in this research can lead to attractive options for embedded system applications.

DD FORM 1 JAN 73 1473

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

Ada programs can be regarded as ensembles of machines, one per program unit (module), which in turn may be mapped directly into corresponding VLSI structures on one or more chips with interconnecting (packet switched or other) communication nets.

Many of the transformation steps, when performed manually, when optimization is not everywhere crucial, and when care is taken to constrain somewhat the structure of the source Ada program, appear to be understood.

The research reported here is part of a five-year plan, the first year of which focuses on "proving" the concepts through a realistic demonstration of methodology for a specific example Ada program (a silicon representation of part or all of the DoD Standard Internet Protocol, IP, initially expressed in Ada.) Since the mapping from Ada to VLSI is seen as a multistep, iterative procedure, considerable effort for the following four and a half years will be the invested in the development and tailoring of intermediate languages and their bridging algorithms (compilers), as needed, and in the development of objective criteria for their use with feedback loops for iterative design.

Implicit in these objectives is the development of a set of hardware structuring paradigms (rewrite rules) whose application can ensure that transformation steps between levels of abstraction in the design process are well structured in order to preserve the integrity and, where possible, the clarity of the original Ada specification. Some paradigms, but of course not all, lead to highly efficient implementations.

Abstract

This report summarizes the second six months of work of the coordinated research project, "Transformation of Ada Programs into Silicon." (The main objectives of this project were outlined and then introduced in depth in the preceding semiannual report.) In the past seven months, work has advanced in three main areas. Expanded summaries of work in these areas (and subareas) are presented:

1. Work on the principal case study of this project: Converting the DoD Internet Protocol to silicon. The full Protocol has been decomposed into three main parts. The part that handles outbound datagrams has been fully specified in Ada and an interesting part of that code has been transformed into an NMOS circuit composite represented in PPL (Path Programmable Logic).

2. A tranformation system is being implemented to map Ada program units into intermediate forms in syntactically correct Ada. These intermediate forms are suitable for input to the transformation system (ASSASSIN) that automates the production of the asynchronous control components of the PPL circuit composites. A theory for synthesizing circuits from system specifications that are more abstract than Ada is also reported.

3. Research and Development on the design, fabrication, and application of PPL (Path Programmable Logic) circuit arrays is reported

   a. The ASSASSIN system which transforms state graphs of state machines expressed in textual form to self-timed PPL programs and composites is operational.

   b. Completion of a PPL simulator (ASYLIM) has been incorporated into the PPL design system.

   c. Design and composite layout of three different PPL test circuits were sent out for fabricaton. The circuits will be used to check a wide variety of PPL cells and supporting circuitry.

   d. A design technique for ICs representing self-timed stored state machines and data path components using the PPL cell set has been developed. The results of the research have produced new PPL macro cells which augment the set of available cells.

1. Summary

This report summarizes the second six months of work of the coordinated research project, "Transformation of Ada Programs into Silicon." Project objectives span a broad and ambitious spectrum (broader than the already broad title implies), hence the term ***coordinated***; this refers tc the fact that, on the one hand, all research within the project is closely related, but that the overall project success is not predicated on close coupling of individual subproject results. The main objectives of this project were outlined and then introduced in depth in the preceding semi-annual report [19]. They are repeated here in more brief and in a somewhat updated form:

1. Develop elements of a tranformation methodology for converting Ada programs or their parts, into VLSI systems. This research includes identifying a sufficient set of transformation rules for mapping program specifications through successive levels of representation, from Ada or related abstract specifications, to integrated circuits.
2. Demonstrate the methodology developed in 1 by manually applying it to a non-trivial example: transforming an Ada-encoded representation of the DoD Standard Internet Protocol [20] (or a significant subset thereof) into NMOS circuitry.
3. Work toward a theory for identifying substructures within Ada programs for which the transformation methodology is pragmatically attractive.
4. Develop specifications for a set of software tools for use in automating the transformation methodology developed in 1.
5. Develop a methodology for testing integrate circuits representing Ada program units and for integrating such circuits into a larger system.

In the past seven months, our work has advanced in three main areas and in several subareas listed below. Expanded summaries of work in these areas are presented in succeeding sections of this report.

1. Work on the principal case study of this project: Converting the DoD Internet Protocol to silicon. The full Protocol has been decomposed into three main parts [18, 13]. The part that handles outbound datagrams has been fully specified in Ada [14] and part of that code has been transformed into an NMOS circuit composite [6].
2. Implementing a tranformation system to map Ada program units into intermediate forms in syntactically correct Ada. These intermediate forms represent <state machine, data path> pairs suitable for input to another transformation system that automates the production of circuit composites [24].
   a. Development of a theory for synthesizing circuits from system specifications that are more abstract than Ada, e.g., axiomatic algebraic specifications or from Ada augmented with ANNA-like specifications that also allow specification of temporal properties. [12, 29, 25, 26]
3. Research and Development on the design, fabrication, and application of PPL (Path Programmable Logic) circuit arrays.
   a. Completion of the transformation system called ASSASSIN, reported in detail elsewhere [7], which transforms state graphs of state machines expressed in textual form to self-timed PPL Programs and composites.
   b. Design and composite layout of three different PPL test circuits called UU20, UU21, and UU23. UU20 is used to check the read-enable flip-flop, the write-enable flip-flop, the asynchronous-clear flip-flop, row pass-transistors, and flip-flop pull-up cells. UU21 checks the Set/Reset flip-flop, the two-wire latch, the inverter cells, the column pass-transistor, and the S, R, 1, and 0 cells. UU23 checks the input and output pad cells. In addition, a test circuit containing several different oscillators and counters has been included for determining performance.

      UU20 and UU21 were sent to MOSIS for the June 4 run, and in July we were informed that, due to some mask problems, none of the circuits were completed. We are still waiting for these parts. In September we decided

to process all three test circuits in our own (HEDCO) laboratory. Problems with mask making equipment have caused delays, however, UU20 and UU21 are expected out of the process line in late November or early December. UU23 should also be processed in December.

c. Completion of a PPL simulator called ASYLIM which has been under development for the past year. (The work was sponsored primarily by a commercial company. The simulator was incorporated into the PPL design system for use in this project. The main characteristics of this simulator are outlined in Section 4 of this report.

d. Development a design technique for ICs representing self-timed stored state machines and data path components using the PPL cell set. (The work was sponsored by a private company.) These techniques have been primarily directed at the design of circuits using a conventional single-rail Four Cycle signalling protocol. The results of the research have produced new PPL macro cells which augment the set of available cells

2. Converting the DoD Internet Protocol to Silicon.

by

Elliott I. Organick and Gary Lindstrom

As mentioned previously [19], our design of the Protocol is based on a decomposition into three submodules: INM_OUT dealing with traffic outbound on a given local net, INM_IN similarly handling inbound traffic, and INM_SRV tying them together and interfacing to the Host(s). We envision one INM_IN and INM_OUT pair of submodules for each local net interface, but only one INM_SRV submodule per Internet Module (INM).

We are following the five-level software development and testing plan discussed in the preceding report. The levels correspond to IP applications in increasingly generalized settings. The plan stipulates testing as each level is reached, rather than as an epilog to the development plan. Testing is to be conducted at several levels, from the physical characteristics of the circuits themselves to the (Ada) semantic behavior of the submodules that have been converted to circuits.

After designing (specifying) the interfaces between the submodules [13, 10], we then selected the INM_OUT (sub)module as the first one to be converted to circuitry. Work toward this objective in the past seven months has been rapid in some respects and slow in others.

The specific and significant accomplishments have been as follows:

1. We have coded the complete INM_OUT submodule in Ada and have succeeded in compiling most of it for execution on the Intel iAPX 432 system except for statements and declarations associated with uses of the Ada rendezvous construct.

   [As later versions of the Intel compiler become available, we expect not only to be able to compile the full module using rendezvous syntax and semantics, but to execute it in this mode as well. In the meantime we are working with a version of the code that simulates each rendezvous via Send/Receive primitives instantiated through use of the Ada generic package mechanism.]

2. The INM_OUT submodule is an Ada package named INM_OUT_Module; it contains three intercommunicating Ada tasks. We are in the process of transforming each of these tasks into PPL circuit composites beginning with the second one listed below:

   a. The main task, named INM_OUT, interfaces with INM_SRV and with LNM_OUT such that a pipeline effect is achieved for speeding datagrams along the outbound data path: Host module —> INM_SRV —> INM_OUT —> LNM_OUT.

   b. An auxiliary (server) task, named Read_Init_Parameters, which obtains from host-related memory the initial parameter values needed to perform datagram transmission. Transformation of this server task, one which is rich in Ada control structures, is essentially completed. A demonstration, showing the process by which we make the transformation to PPL circuit composite was given in June, 1982 during a DARPA review of our project. That demonstration was based on a preliminary version of the Ada task, which has now been updated. The composite produced for the current version of the task is more interesting and is apt to resemble more closely the one we eventually will consider the final version.

   c. An auxiliary task named Translate_TOS_Task, which operates in parallel with INM_OUT, the main task, by translatiing type-of-service information from host-level to local-net level encoding.

3. As just mentioned, the task Read_Init_Parameters has now been converted semi-automatically to PPL circuit composites in NMOS. The conversion into PPL composite form is discussed in part in a new paper by Carter, to be presented at a DARPA-sponsored meeting at Stanford, on November 5 and in part below. Carter's paper focuses primarily on the technology for converting the control structure portion of the Ada task into the self-timed control unit of the

corresponding circuit.

In this report we make some observations on the overall structure of Read_ Init_ Parameters and on some of its subtle details. We also comment on some of the steps we traversed in arriving at this version of the task. A copy of the body part for the present version of this Ada task is to be found in the Appendix.

[The complete Ada specification of the INM _ OUT submodule, which includes this task is given in a separate report [14]. A reader of the Appendix version only is expected to imagine how the task Read_ Init_ Parameters interfaces with the remainder of the entire submodule. A reader of the separate report is treated to a "road map" of the full Ada structure of the INM _ OUT submodule which helps to understand our overall design.]

4. As a prelude to testing hardware versons of Ada pargram units and in support of our work in specifying subsystems in Ada and then simulating them, we installed, made operational, and have begun using a complete Intel 432 Cross Development System. This system includes an Ada cross compiler for a large subset of Ada and a 432 multiprocessor system consisting of two regular and two interface processors. We expect to receive from Intel a compiler that includes full tasking by the end of calendar 1982 and an equally complete resident compiler approximately a year later. We have also gained hands-on familiarity with a number of the 432 System's operating system features.

2.1. Interesting aspects of Read_ Init_ Parameters

The structure of Read_ Init_ Parameters includes a number of typical and interesting features of Ada tasks both from the point of view of inter-task communication and intra-task body structure.

- Inter-task communication. The task includes nested accept statements both of which have both in-bound and out-bound parameters. There accept statements are implemented using simple request/acknowledge protocols.
- Intra-task computation. The task body includes a rich nested loop structure and one nested block defining local variables whose ranges are determined dynamically. The loops include the infinite outermost loop of the task, familiar "for" loops with fixed upper bounds, and indefinite loops escapes from which are based on "exit when" clauses. As we have expected all along, all of these Ada control structure forms map in a straightforward way to corresponding control structures at the state machine level and thence to PPL circuits.

The data path of Read_ Init_ Parameters includes several variables which are represented in the hardware as registers or counters. One array variable is represented as a RAM to represent a map from type-of-service encoded at the host level to type-of-service encoded at the local net level. [The size of this RAM, which is never apt to be very large in any case, is limited to four-octets (for a 2 by 2 array) in our demonstration implementation. Most of the above variables are shared with the other two tasks of the submodule; that is, they are declared local to the containing package, INM _ OUT _ Module, however we perceive no difficulty in achieving mutually exclusive access.

The one variable that is local to the entire server task does not and is not represented in hardware as a storage element. Variables used locally for loop control are represented as hardware counters and/or registers, but some sharing is achieved where there is no chance for conflict.

Although the transformation to the Ada code to the "engine level", i.e., to representation as a (control unit, data path) pair, has been done by hand, the transformation research reported in the next section has included consideration of each of the "hand-made" mapping steps in this particular exercise.

2.2. Arithmetic processing

That we have encountered so little trouble performing the mapping for this task is partially explained by the fact that the task involves only trivial arithmetic processing. (Indeed, the entire INM_OUT_Module involves only minor arithmetic processing.) At this stage of our research we are glad this is the case as we consider it important to determine first what new challenges, if any, must be met for achieving asynchronous control.

2.3. On going and future related work

Now that this part of the research is essentially complete, including the development of the ideas embodied in ASSASSIN, we expect to be concentrating next on such challenges as the application of the same or related asynchronous design principles to arithmetic processing. Also included in our agenda is research intended to help us automate the mapping of data path storage components, identified in the transformation from Ada program units, into PPL circuits coupled to their controls.

3. A Transformation System: Theory and Implementation

by

P.A. Subrahmanyam

We have made substantial progress along two directions: implementation of a prototype transformation system and further development of a conceptual/theoretical basis to support the design of integrated software–hardware systems. We outline the major contributions below, with appropriate pointers to references that contain more detailed discussions.

3.1. Systems Implementation

- —A set of tools to support experimentation with Ada–to–Silicon transformations has been implemented, and runs on the TOPS–20. The system has been ported to the VAX–750, and an initial version has been installed. This porting proved to be a major job (and problem) due to unstated incompatibilities between INTERLISP–20 and INTERLISP–VAX. Further debugging and testing of the Vax version will be done when the experimentation is moved completely over to the Vax. (Given the needed personnel, we expect this to be carried out over the next year, when our address space requirements force us to move over to the Vax).
- —An initial set of transformation routines has been implemented and is being augmented so as to handle additional syntactic constructs in Ada. This set of programs is intended to aid in the interactive generation of the target hardware description in a symbolic representation. Details of the current status of this work are reported in [24].

3.2. Conceptual/Theoretical Basis for Transformation

- —A unified theoretical framework to support a broad spectrum of the VLSI design process has been introduced in [29], which is currently available in the form of the draft of a research monograph. This monograph introduces an algebraic framework to aid in the synthesis and verification of special purpose VLSI systems, proceeding from high level specifications. It allows for abstract specifications of the syntax, semantics, temporal and performance requirements particular to a given problem. The characteristics of the environment in which the system is embedded can also be specified and are used in the synthesis process. In addition, the framework allows several of the constructs in existing languages to be modelled, including nondeterminism, concurrency, and data/demand driven evaluation. This allows the infrastructure to be (1) applied to situations wherein the problem "specification" is in the form of a program in a conventional high level language and (2) used to model the lower level synchronous/asynchronous nature of implementations. Topology and circuit layout geometry can also be expressed by using the algebraic primitives available.
- —Annotations to Ada have been proposed to aid the abstract specification of temporal properties of systems and desired performance requirements [25, 28, 12].
- —Transformation methods to apply the theory in the context of Ada to obtain systolic implementations are detailed [27, 24].
- —An algebraic modelling of weak conditions to be met by asynchronous circuits has been done — the resulting model is very simple, and the conditions concise and intuitive [26].

Following a discussion of the specification and synthesis methods, illustrations are given in [29] that demonstrate the use of the proposed theoretical basis in synthesizing various classes of algorithms. It is shown how (families of) systolic algorithms may be obtained as a special case. Methods for proving the correctness of implementations are presented and illustrated with examples. The concept of the propagation of computational loci arises naturally in course of the development, and serves to generalize the commonly used notion of a "wavefront" of computation for 2–dimensional architectures. Automatable design aids based on the proposed algebraic basis are delineated. Finally, it is shown how MOS circuits can be modelled using the primitives available, and the algebraic derivation of Bryant's simulation algorithm used in MOSSIM II is illustrated in this context.

3.2.1. Interface With Diana

Most of our transformation tools use the parse tree representation of a program as the primary data structure they work with. We have in mind the long term objective of being able to interface with the tools that are designed to operate on Ada program parse trees, and that being developed by the Ada community at large (and in particular the DARPA community). To this end, we have been interacting (to a limited extent) with the Diana group (primarily at Tartan Laboratories).

3.3. Some Remarks on System Implementation Issues

While we are continuing work on the current version of the transformation system (in Interlisp, and on the Vax and DEC–20), it has become clear that there are two major deficiencies that need to be remedied sooner or later. These are (1) unsuitability of the current parse tree interface (and parser generator) for several of the transformation routines themselves; and (2) (lack of) speed: this is due to the slowness of Interlisp on the Vax (compounded, of course, by the fact that we are working with non–trivial pieces of software).

To solve the first problem, it is necessary to redesign the parser generator (which has been imported from ISI [31]). However, since the other tools (particularly the syntax directed editor generator and pattern matching system) and the history list mechanism are all very much inter–related and quite deeply ingrained in the system, there is a substantial software development effort involved in doing this. Currently, we have neither the equipment nor the man–power to support such an effort. We envision the redesign being more profitably done using a newer generation of Lisp (e.g. PSL, CommonLisp) for efficiency reasons, and run on personal machines, rather than on a Vax like machine. In the interim, however, the response of the extant version of our system can also benefit greatly from being run on an Interlisp—supporting machine, e.g., the Dorado/Dolphin. Having access to such systems would obviously result in greatly improved programmer productivity.

4. PPL Design Activities

by

Kent F. Smith, Brent Nelson, Tony Carter, and Alan Hayes

A system for the design of integrated circuits using a methodology known as Path Programmable Logic (PPL) has been developed by the Utah VLSI Group. This work has been sponsored in part by the DARPA contract and by contracts with other government agencies and in part by support from several independent companies. The system addresses the complete design cycle including initial logic design, circuit layout, simulation, electrical checking, and pattern generator tape preparation. It includes: (1) symbolic layout programs to facilitate the placement of the symbols on the grid, (2) a simulator patterned after switch-level simulators but specifically tailored for use on PPL, (3) a checker program for cell placement verification and DC circuit loading checking, and (4) a common database for design representation.

4.1. PPL Design Characteristics

The characteristics of design using the PPL methodology include:

1. IC design is performed by placing small circuit modules which can be represented with logic symbols on a grid representing the integrated circuit. When the grid is completely populated, it is both the logical representation and the topological layout of the circuit. Efficient design changes can be made as a result of this design methodology because the designer has simultaneous perception of the circuit function and the circuit topology.
2. The circuit modules have predefined schematic and composite representations. They are custom designed to optimize performance and size for any specific integrated circuit process. Design Rule Checking (DRC) is performed on the module and thus it is not necessary to do DRC on the overall circuit since it is simply a collection of circuit modules.
3. A complete circuit can be designed in PPL and no custom design is required. The pads and the interconnect can also be made by the placement of PPL cells on the grid. All interconnections between modules are there by default. The designer only places breaks to remove connections rather than to add them.
4. Hierarchical design is possible by custom design of macros which are collections of PPL cells put together to perform specified functions. These macros cells can have custom physical shapes to conform to specific space requirements.
5. Simulation and checking are easily accomplished, eliminating the need for very difficult and time-consuming operations. The only elements manipulated are symbols rather than transistors or rectangles which must be checked in systems that design at the transistor level.

4.2. The Analogy Between the PPL Design and a Computer Program

There is an analogy between the development of the PPL design methodology and programming languages. The 1's and 0's which were used in early machine language computer programming are analogous to the rectangles which are used in the custom layout of integrated circuits. Placing transistors on a composite might be thought of as being analogous to writing machine language code in hexidecimal since we are still placing rectangles on a grid in shorthand form. The PPL design methodology is analogous to writing programs in assembly language where mnemonics are used to represent specific collections of transistors (functions). This PPL design methodology is still very dependent upon the specific technology which it is designed in. This is similar to the way that assembly language is machine-dependent.

The analogy between the development of computer programs and the PPL methodology can be carried even further with the compilation of high level circuit description languages to integrated circuit layouts (silicon compilers). The high level descriptions of the integrated circuit are machine independent and are compiled directly to a specific PPL cell set designed in a particular technology. To date there have been cell sets done in NMOS [21], CMOS [22], and I2L [23]. An example of such a silicon compiler is ASSASSIN [7] which is currently in use at the University of Utah.

4.3. Design Time vs. Integrated Circuit Area

The main disadvantage of PPL design methodology is that it will probably result in circuits which are larger than completely custom-designed circuits. Previous work done by the VLSI group at the University of Utah has compared some custom designs to some PPL designs. This gives insight into the tradeoffs which exist between the two techniques. A circuit known as the Utah Serial Cordic Machine (USCM) was designed under a contract with Wright Patterson AFB for the VHSIC program [3, 4, 5] using both custom design techniques and the PPL Design Methodology. The USCM was constructed using an implementation similar to the shift-register scheme proposed by Volder [30].

The USCM was implemented using a CMOS PPL cell set. Its design time and chip area were compared to those for an equivalent custom NMOS design done at Boeing Aerospace Corp. The entire CMOS PPL chip was designed and simulated in approximately eight man days, compared to approximately eighty man days for the NMOS custom design. The CMOS PPL design was 19 percent larger than the custom NMOS design. While these figures may not be an accurate reflection of the variables which enter into design time measurements, they are indicators that PPL designs require significantly less design time than do equivalent custom designs and result in chips which are not significantly larger in area.

This favorable reduction in design time can be attributed to several factors: (1) The designer has concurrent perception of logical function and layout. Thus, he can immediately see when the logic function being implemented does not fit in well with the rest of the circuit. The logic design is made as the composite is drawn. This eliminates the need for separate composite layout/logic design stages. (2) The higher level symbolic notation allows the designer to manipulate very complex logical elements in an efficient manner. It is, for example, not necessary to trace a complex series of logic gates to determine the function of the circuit because the symbolic notation is easily read and interpreted. In addition, the symbolic notation can be directly simulated and does not require the extraction of the transistor-level circuit from the composite.

Past experience would indicate that the area penalty incurred by the PPL design methodology will eventually disappear as more sophsticated design tools are developed. This is again analogous to the development of compilers. It is well known that, as expertise in compiler writing improved, the gap between hand-coded and compiler-produced object code size became negligible. Some of the techniques being developed for compaction of integrated circuit layouts will be used to close the current gap between the area required for custom designs and automatically generated PPL layouts.

4.4. The Utah PPL Design System

In addition to the development of the PPL as a hardware implementation methodology described above, the other major thrust of research here at Utah has been in developing software tools for PPL design. The goals of this software research have included the following: (1) Finding ways to exploit the symbolic nature and representation of a PPL design to reduce design complexity. (2) Development of CAD tools around conventional computer hardware, which would allow designers to work from remote workstations. (3) Creation of a complete system to be used by the IC design community here at Utah.

An integral part of the design system is a Computer Vision CADDS2/VLSI Designer System. It is used to do the composite layout of the individual PPL cells, placement of the individual cells on a grid to form a circuit, connecting the circuit to pads, adding scribe lanes, and generating a PG tape. Although we have relied heavily on this machine in the initial development of the system, in its absence all of the functions it performs could be done with other tools (the Cal-Tech Software Package for example).

The other part of the design system is built around a DECSystem-20. A silicon compiler for finite state machines (FSM), a symbolic layout system, a simulator and cell placement checker, and a compaction program all reside there. The transfer of designs between the Computer Vision machine (CV) and the DECSystem-20 is done using a mag tape written in Computer Vision External Database format. The combination of these two computers gives the system the power of the CV's IC layout features combined with the computing power of a mainframe.

Each PPL cell used in the system has three representations. The composites of the cells are designed so that they fit together by virture of their being placed adjacent to each other on the grid. A schematic representation of each cell is created for reference. A graphical representation is also created which is used by the designer as he uses the cells to form larger circuits.

4.5. Presently Existing Circuit Layout Tools

The placement of the PPL cells on the grid to form a circuit can be done using either the Computer Vision machine or one of several programs on the Utah DECSystem-20. The program used for cell placement on the DECSystem-20 is known as SLED (Structured Logic Editor) [15]. In SLED, the PPL design is represented as an array of cell symbols which are then edited. With the SLED editor, a simple CRT terminal and modem is all that is needed for circuit design but at the expense of more cryptic graphical representations of the individual PPL cells than those found on the Computer Vision machine. In general, the ability to use SLED from a remote terminal outweighs this limitation. Advanced editors are now being designed to run on a CRT terminal that will overcome some of the graphical limitations of SLED.

SLED was designed to be similar to a screen-oriented text editor. In fact, the commands in SLED are the same as the equivalent commands in EMACS [8], a popular screen-oriented text editor. Cursor movement is possible in any of the four directions, and regions (windows) can be marked and then named, deleted, replicated, or written to a disk file. Conventional text editors, however, only allow for scrolling and windowing in the vertical direction (lines longer than the width of the screen are wrapped around). In SLED, scrolling and windowing are possible in both directions. Thus, an array with 300 columns and 300 rows can be displayed and edited using SLED without screen wrap-around. The effect is that the user has an 80X24 window which can be moved around the array.

Circuit layout can also be accomplished using a first-generation silicon compiler. Compilaton of Ada language modules to circuits is accomplished using the program named ASSASSIN [7]. This program takes as its input a textual description of the operation of a control unit (Finite State Machine) and from it generates a PPL layout implementing the control unit.

4.6. Circuit Simulation and Electrical Checking

Simulation of the PPL design is essential before actual fabrication. An important part of the design system is a simulator (ASYLIM) which can do simulation of the PPL. Because the PPL cells are simulated and checked individually at the transient level when the cell set is designed, the complete circuit made up of PPL cells can be simulated at a switch or gate level. ASYLIM [16, 17] reads the circuit database written in Computer Vision External Database format. Thus, the actual design can be simulated rather than a logic equivalent.

ASYLIM is similar to other recently developed MOS simulators in that it uses a switch model. However, the development of a simulator for PPL has shown [17] that a special purpose simulator was required in order to preserve the user's abstract view of the circuit. The input format to existing simulators is typically given in the form of a table or listing of transistors and nodes. To preserve the user's abstract view of the circuit it was necessary to design a simulator for PPL where the elements in the simulator correspond to those in the PPL cell set. During the interactive debugging phase of the simulation of a circuit, the user can then refer to circuit elements by their **position** in the PPL array. An added feature of the PPL simulator is that the information stored in the simulator's internal representation of the circuit interconnect structure can be used for additional circuit checking unique to the PPL methodology. The end result is that ASYLIM is similar to conventional switch-level simulators but with an extensive user-interface that allows the user to work with the circuit at the symbolic PPL level, the same level he uses when designing.

ASYLIM makes use of six-valued logic and uses a unit-delay timing model [1, 2]. The underlying circuit model primitives are switches but with extensions to allow for the simulation of certain entities as gates (flip flops and latches). It has been shown that the unit-delay model is adequate provided the circuit is free from races. Thus it can be used to model the sequence of circuit activity [2].

An additional advantage of using ASYLIM over other simulators is that it contains an extensive interactive circuit debugger. The features of this debugger allow the user to view the circuit interconnect structure as constructed by the simulator. This is displayed in a readable format that allows the user to quickly compare the simulator's interpretation of the circuit element interconnections and the intended design. This comparison uncovers most design errors relatively quickly. In addition, the simulator performs a pre-simulation plausibility check on the circuit's nodal structure. This feature (the idea borrowed from Bryant's MOSSIM [2] enables the user to find a large percentage of the design errors without ever going to the expense of an actual simulation. This check identifies nodes with fanout but no inputs, inputs but no fanout, no path to either power or ground, or multiple pullup loads.

While a logic or switch-level simulation can provide an invaluable service in verifying the logic design, there are many features of a design that do not show up in a simulation run. For example, the ground node may be specified as an input to a transistor in a diagram but it requires an explicit check on the layout to ensure that ground actually has been routed to that device. In PPL design, these types of electrical (non-logic) entities are included in the design using special cells. For instance, the power bussing structure is included by placing power and ground buss cells around the circuit perimeter. In addition, other cells, like row and column loads, are usually left of out of logic diagrams but must be included for the circuit's correct operation. ASYLIM checks for these cells as a part of its operation.

4.7. Self Timed IC Design with PPL's

Another activity which has been funded by a private company and is of importance in the development of the PPL methodology is the design of self-timed modules using the PPL cell set. The work is based on techniques developed earlier [9] for realizing self-timed stored state sequential circuits. The original investigations were applied to off-the-shelf SSI parts. The present investigations are for the transfer of those ideas to large collections (macros) of PPL cells for use in the design of self timed systems to be contained on single integrated circuits. The investigations have led to further development of the PPL cell set to include methods for self timed circuits [11].

This research has resulted in a design discipline for self-timed stored state machines which has been develcped using a conventional single rail Four Cycle signalling protocol. (State descriptions are encoded in PLAs represented in PPL.) The discipline differs from that used by Carter [7] which uses a technique known as a "one hot" scheme. The approach used for realizing the self timed stored state machines is based on two key developments: (1) A novel clocking circuit that generates a non-overlapping two phase clock cycle for an arbitrary size register, where the duration of the phi 1 phase of the cycle is automatically adjusted to the register size, and (2) A layout discipline for the folded PLA holding the state table, which guarantees that the inputs to the state register will be valid at the time that the clock cycle occurs.

The method depends on certain properties of the NMOS PPL cell set, i.e. that row and clock wires are polysilicon, and that registers are formed by locating flip-flop cells such that their clock lines are serially connected. This method offers a designer the advantage that he need not concern himself with the timing details of a state machine design in order to assure that it will work. Assuming that the state table realized by the PLA is correct, that the rows and columns of the design are properly loaded, and that the proper interconnections have been made (all of which can be verified with the PPL simulator [17]), the designer can be assured of correct operation of the state machine. The principle disadvantage of the method is the overhead of the clocking circuit which must be associated with each state machine.

In addition to the self–timed state machine design, the described design discipline [11] has been applied to several interesting types of self–timed data–path modules, for example multi–bit latches and ripple–carry counters.

4.8. Future CAD Tools for the PPL Design Methodology

Our operational design tools should be enhanced. The following agenda lists the tools we have identified as being an important part of a design system for this methodology and which we plan to develop:

1. A Relational PPL Database Management System — This will allow the same software tools such as the editor and simulator to be used on PPL designs done using any specified integrated circuit technology such as NMOS, CMOS, I2L, and GaAs. In addition, it will provide a standard interface between the various CAD programs.
2. A Symbolic, Interactive, PPL Editor — this editor will be used to create a symbolic representation of a PPL circuit. It will be used interactively by a designer for the semi–automatic placing of PPL cells on the PPL grid. Because of the symbolic nature of PPL, many of the mundane design tasks can be automatically performed by the editor, leaving the designer free to concentrate on logical design. The editor will use either tablet or keyboard entry with simultaneous graphical representation of both the logic description and the circuit topology.
3. Minimization of PPL programs — Development of a compaction program for compressing a PPL design by rearranging its symbolic description. Such a program will use heuristically driven artificial intelligence techniques to arrive at a near–optimal solution to the minimization problem. This tool will give us the capability of doing loosely packed PPL designs which can then be automatically compressed. This is a unique feature of the PPL design methodology and can be accomplished because of the symbolic nature of the PPL.
4. Predefined Structured Logic Blocks — We are persuaded that circuits that already contain large blocks of non–PPL structured logic should be designed using similar techniques to those presently used for the design of such blocks. For instance, if a random access memory (RAM) is required in a circuit, it is more efficient, both from a performance as well as a topological standpoint, to actually do a custom layout of the RAM. The PPL cell set can be extended to include very elementary cells from which macro cells can be developed for any specific implementation of a RAM. Components generated by such an implementation, although not strictly PPLs, would be compatible with their PPL neighbors. A list of of structures we expect to implement as macros includes:

```
nxm ram
nxm rom
n-bit ripple adder
n bit fast adder
n-bit priority encoder
n-bit register
nxm multiplier
n-bit comparator
n-bit synch counter
n-bit ripple counter
n-bit by m:1 MUX
```

4.9. Observations

Our research thus far has demonstrated the usefulness of the PPL methodology as a higher level design technique for hardware analogous to the use of assembly language for computer programming. The analogy has been extended by the introduction of ASSASSIN, a first–generation silicon compiler for speed–independent finite state machines.

Our design system has proven useful for doing actual design of a variety of integrated circuits. It has reduced design times required by an order of magnitude. Resultant designs are easily simulated and corrected due to their symbolic representation. System designers with little or no direct experience with integrated circuit design can do actual IC layout.

5. Project Bibliography of Papers, Reports and Theses

This section contains a cumulative list of the papers, reports and theses regarded as direct or indirect "products" of this Project. Subsequent semiannual technical reports will contain updated versions of the list given here.

[1] Carter, T.M.
ASSASSIN: An Assembly, Specification and Analysis System for Speed-Independent Control-Unit Design in Integrated Circuits Using PPL.
Master's thesis, University of Utah, Department of Computer Science, June, 1982.

[2] Carter, T.M.
ASSASSIN: A CAD System for Self-Timed Control-Unit Design.
Technical Report UTEC-82-101, University of Utah, October, 1982.

[3] Drenan, L.A.
On Transforming Ada to Silicon.
Master's thesis, University of Utah, Department of Computer Science, August, 1982.

[4] Drenan, L.A., Organick, E.I.
Ada to Silicon Trnsformations: The Outline of a Method.
Technical Report UTEC-82-016, University of Utah, Dept. of Computer Science, Sept, 1982.

[5] Hayes, A.B.
Self-Timed IC Designs with PPL's.
October, 1982.
Paper submitted for 1983 Cal Tech VLSI Conference.

[6] Nelson, B.E.
ASYLIM User's Manual
1982.

[7] Nelson, B.E.
ASYLIM: A Simulation and Placement Checking System for Path-Programmable Logic Integrated Circuits.
Master's thesis, University of Utah, Department of Computer Science, October, 1982.

[8] Organick, E.I., and Lindstrom, G.
Mapping high-order language units into VLSI structures.
In *Proc. COMPCON 82*, pages 15-18. IEEE, Feb., 1982.

[9] Organick, E.I., Carter, T., Lindstrom, G., Smith, K. F., Subrahmanyam, P.A.
Transformation of Ada Programs into Silicon. SemiAnnual Technical Report.
Technical Report UTEC-82-020, University of Utah, March, 1982.

[10] Organick, E.I., Carter, T.M., Hayes, A.B., Nelson, B.E., Lindstrom, G., Smith, K., Subrahmanyam, P.A.
Transformation of Ada Programs into Silicon. Second SemiAnnual Technical Report (to appear).
Technical Report UTEC-82-103, University of Utah, November, 1982.

[11] Ramachandran, R.
A Complexity Computation Package for Data Type Implementations.
Master's thesis, University of Utah, Department of Computer Science, June, 1982.

[12] Subrahmanyam, P.A.
From Anna+ to Ada: Automating the Synthesis of Ada Package and Task Bodies.
Technical Report Internal Report, University of Utah, March, 1982.

[13] Purushothaman.S, and Subrahmanyam, P.A.
An Algebraic Model of Seitz's Weak Conditions for Self Timed Systems.
Technical Report UTEC # 82-066, University of Utah, October, 1982.

[14] Subrahmanyam, P.A.
Language Issues in Transformation Systems (to appear).
Technical Report UTEC # 82–069, University of Utah, November, 1982.

[15] Subrahmanyam, P.A. and Rajopadhye, S.
Automated Design of VLSI Architectures: Some Preliminary Explorations.
Technical Report UTEC # 82–067, University of Utah, October (Revised), 1982.

[16] Subrahmanyam, P.A.
A Theoretical Basis for the Synthesis and Verification of Systolic Designs.
Technical Report Internal Report, Dept. of Computer Science, University of Utah, June, 1982.

[17] Subrahmanyam, P.A.
On Automating the Computation of Approximate, Concrete, and Asymptotic Complexity Measures of VLSI Designs (to appear).
Technical Report UTEC–82–095, Dept. of Computer Science, University of Utah, November, 1982.

[18] Subrahmanyam, P.A.
Automatable Paradigms for Software–Hardware Design: Language Issues.
In J.Rader (editor), *IEEE Workshop on VLSI and Software Engineering*. IEEE, October, 1982.
Also available as University of Utah Technical Report UTEC–82–096, September 1982.

[19] P.A. Subrahmanyam.
An Automatic/Interactive Software Development System: Formal Basis and Design.
North–Holland, Amsterdam, 1982, .

[20] Subrahmanyam, P.A.
Abstractions to Silicon: A New Design Paradigm for Special Purpose VLSI Systems.
Technical Report UTEC # 82–065, University of Utah, January, 1981 (Revised May 1982).
Submitted for Publication to TOCS.

[21] Subrahmanyam, P.A.
An Algebraic Basis for VLSI Design.
Draft of a Research Monograph, April 1982, 120 pp. Available from the Department of Computer Science, University of Utah.

6. Appendix

```
-----------------------------------------------------------------------
--                                                                   --
--                      Ada-to-Silicon Project                       --
--                       University of Utah:                         --
--                                                                   --
--              DoD Internet Protocol INM_OUT submodule              --
--                                                                   --
--         Ada code for the body of task Read_Init_Parameters        --
--                    Version of October 25, 1982                    --
-----------------------------------------------------------------------

separate (Inm_Out_Module)


task body Read_Init_Parameters is

  -- Accessed globals:
  -------------------
  -- number_of_local_net_types_of_service:           octet_type
  -- local_net_type_of_service_table_row_size:       octet_type
  -- tos_table:                                      octet_buffer_type

  -- Renamed task entry:
  ---------------------
     -- The package Memory_Module containing the task Memory holds
     -- to-be-sent datagrams as well as initialization parameters
     -- needed by INM_OUT.

  procedure Memory_request(
      request_type_formal:          memory_request_type;
                                       -- Load_address or receive_datum_octet.
      chunk_of_address_formal:      chunk_of_address_type;
                                       -- Don't care when request_type_formal
                                       -- receive_datum_octet.
      octet_formal:             out octet_type)
                                       -- Don't care when load_address.
    renames Memory.Request;

  -- Local variable declaration:
  -----------------------------
  -- The following variable is commented out. It appeared only in the
  -- "high-level" used to read in the TOS table.  See below.
  --  number_of_tos_table_octets: integer range 2 .. max_tos_table_size - 1;
  octet_register:                 octet_type;

begin
  loop
    accept Go(
        init_num_formal:            bit4;                 -- For Carter's paper
                                                          -- only; otherwise bit3
        response:           out out_response)
      do
        response := sent_ok;                              -- Also means init_ok.

        -- Get from the server all of the addr_chunks needed to form the base
        -- address in memory that holds the initialization parameters and
        -- sends these chunks to the Memory module.
        for index in 1 .. init_num_formal
        loop
          accept Srv_req(                                -- Get next address
                                                          -- chunk from the
                                                          -- Server Module.
              server_command_datum:      srv_command;
              response_to_server:    out out_response)
          do
            Memory_request(                               -- Put chunk out to the
                                                          -- Memory module.
            request_type_formal      => load_address,
            chunk_of_address_formal =>
                                    Convert_srv_command_to_chunk_of_address
                                        (server_command_datum),
            octet_formal             => dont_care_octet);
        end Srv_req;
      end loop;

      -- Get the 6 individual initialization parameters (contained in the
      -- next 8 octets received) from the Memory Module.
      for index in 1 .. 8
      loop

        Memory_request(
            request_type_formal      => receive_datum_octet,
            chunk_of_address_formal => dont_care_X_datum,
            octet_formal             => octet_register);

        case index is
          when 1 => inm_max_packet.lo          := octet_register;
          when 2 => inm_max_packet.hi          := octet_register;
          when 3 => inm_address_length         := octet_register;
          when 4 => inm_time_out.lo            := octet_register;
          when 5 => inm_time_out.hi            := octet_register;
          when 6 => ack_type                   := octet_register;
          when 7 => local_net_type_of_service_table_row_size
                                               := octet_register;
          when 8 => number_of_local_net_types_of_service
                                               := octet_register;
        end case;
      end loop;

      -- Convert the local net timeout into milliseconds.?
      -- time_out_in_milliseconds := inm_time_out / 1000.0;
                                          -- Left-hand side variable declared
                                          -- in Inm_Out_Module. Value is used
                                          -- later in Do_send procedure.
                                          -- Note: Davis never did this in
                                          -- his design. Is this step needed?
                                          -- No! We don't need this step
                                          -- since the  quotient can be
                                          -- approximated by a div by 2**10
                                          -- in the event we need to
                                          -- represent milliseconds.

      -- Read in type of service translation table.

        --   The following code in comments is replaced below by a
        --   "lower-level" version that closely reflects the hardware
        --   implementation chosen in which we eliminate the need for
        --   for a multiplier.

--      number_of_tos_table_octets := local_net_type_of_service_table_row_size
--                                    * number_of_local_net_types_of_service;

--      -- Check to see if required table size exceeds maximum
--      if  number_of_tos_table_octets > max_tos_table_size  then
--        response := bad_srv_command;
--        return;
--      end if;

--      for index in 1 .. number_of_tos_table_octets
--      loop

--        Memory_request(
--            request_type_formal      => receive_datum_octet,
--            chunk_of_address_formal => dont_care_X_datum,
--            octet_formal             => tos_table(index));
--      end loop;
```

```
        declare
          row_number: integer range 0 .. number_of_local_net_types_of service;
          col_number: integer range 0 ..
                                  local_net_type_of_service_table_row_size;
          index:      integer range 0 ..
                                  number_of_local_net_types_of service
                                  * local_net_type_of_service_table_row_size
                                  := 0;
        begin
          row_number := 0;
          loop                    -- Outer loop reads all rows of TOS table.
            col_number := 0;
            loop                  -- Inner loop reads in one row of TOS table.
              Memory_request(
                  request_type_formal      => receive_datum_octet,
                  chunk_of_address_formal  => dont_care_X_datum,
                  octet_formal             => tos_table(index));

              col_number := col_number + 1;
              exit when col_number = local_net_type_of_service_table_row_size;

              index := index + 1;
              if  index > max_tos_table_size  then
                response := bad_srv_command;
                return;           -- Exit the current accept statement.
              end if;
            end loop;             -- End inner loop.

            row_number := row_number + 1;
            exit when row_number  = number_of_local_net_types_of_service;
          end loop;               -- End outer loop.
        end;                      -- End declare block.

      end Go;                     -- End of init processing.

    end loop;                     -- End of outer-most (inifinite)
                                  -- loop.

end Read_Init_Parameters;
```

References

[1] R. E. Bryant.
Logic Simulation of MOS LSI.
PhD Disseration Proposal, Massachusetts Institute of Technology, January, 1980.

[2] R. E. Bryant.
A Switch-Level Simulation Model for Integrated Logic Circuits.
PhD thesis, Massachusetts Institute of Technology, 1981.

[3] T. M. Carter and K. F. Smith.
Applications of Logic Arrays in VHSIC Design.
March, 1981.
Quarterly Technical Report #2 from the VLSI Research Group at the University of Utah, Department of Computer Science, to Boeing Aerospace Company.

[4] T. M. Carter; K. F. Smith; C. E. Hunt; and W. L. Howard.
Applications of Logic Arrays in VHSIC Design.
June, 1981.
Quarterly Technical Report #3 from the VLSI Research Group at the University of Utah, Department of Computer Science, to Boeing Aerospace Corporation.

[5] T. M. Carter; K. F. Smith; C. E. Hunt; and B. E. Nelson.
Applications of Logic Arrays in VHSIC Design.
September, 1981.
Quarterly Technical Report #4 from the VLSI Research Group at the University of Utah, Department of Computer Science, to Boeing Aerospace Corporation.

[6] Carter, T.M.
ASSASSIN: A CAD System for Self-Timed Control-Unit Design.
Technical Report UTEC-82-101, University of Utah, October, 1982.

[7] T. M. Carter.
ASSASSIN: An Assembly, Specification and Analysis System for Speed-Independent Control-Unit Design in Integrated Circuits Using PPL.
Master's thesis, Department of Computer Science, University of Utah, June, 1982.

[8] Richard M. Stallman.
EMACS Manual for TWENEX Users
Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1980.

[9] A. B. Hayes.
Stored State Asynchronous Sequential Circuits.
IEEE Transactions on Computers C-30(8):596-600, August, 1981.

[10] Alan B. Hayes.
High-level Logic Design of the DoD INM-OUT Module.
April, 1982.

[11] Hayes, A.B.
Self-Timed IC Designs with PPL's.
October, 1982.
Paper submitted for 1983 Cal Tech VLSI Conference.

[12] Krieg-Bruckner, B., Luckham, D.C., von Henke, F.W., Owe, O.
(Draft) Reference Manual for Anna, A language for Annotating Ada Programs.
Unpublished, Reviewer's Copy, October 1982.

[13] Lindstrom, G.
Internet Protocol Case Study: Background and Initial Design.
May, 1982.

[14] Lindstrom, G., Organick, E.I., Klass, D., Maloney, M.
Ada Specifications for the DoD Internet Protocol: The INM_OUT Submodule, Report No. 1.
Technical Report, Department of Computer Science, University of Utah, November, 1982.

[15] Brent E. Nelson.
SLED User's Manual
1982.
Department of Computer Science, University of Utah.

[16] Brent E. Nelson.
ASYLIM User's Manual
1982.
Department of Computer Science, University of Utah.

[17] Brent E. Nelson.
ASYLIM: A Simulation and Placement Checking System for Path-Programmable Logic Integrated Circuits.
Master's thesis, University of Utah, October, 1982.

[18] Organick, E. I., and Lindstrom, G.
Mapping high-order language units into VLSI structures.
In *Proc. COMPCON 82*, pages 15-18. IEEE, Feb., 1982.

[19] Organick, E.I., Carter, T.M., Lindstrom, G., Smith, K.F., Subrahmanyam, P.A.
Transformation of Ada Programs into Silicon. SemiAnnual Technical Report.
Technical Report UTEC-82-020, University of Utah, March, 1982.

[20] Postel, Jon: editor.
Internet Protocol: DARPA Internet Program, Protocol Specification.
Technical Report RFC 791, Information Sciences Institute, USC, Sept., 1981.

[21] K. F. Smith.
Implementation of SLA's in NMOS Technology.
In *Proceedings of the VLSI 81 International Conference, Edinburgh, UK*, pages 247-256. August, 1981.

[22] K. F. Smith; T. M. Carter; and C. E. Hunt.
The CMOS SLA and SLA Program Structures.
In H. T. Kung; B. Sproull; and G. Steele (editor), *Proceedings of the 1981 CMU Conference on VLSI Systems and Computatons*, pages 396-407. Computer Science Department, Carnegie-Mellon University, Computer Science Press, October, 1981.

[23] K. F. Smith.
Design of Stored Logic Arrays in I2L.
In *Proceedings of the 1981 IEEE International Symposium on Circuits and Systems*, pages 105-110. IEEE Circuits and Systems Society, April, 1981.
IEEE Catalog No. 81CH1635-2.

[24] Subrahmanyam, P.A. and Rajopadhye, S.
Automated Design of VLSI Architectures: Some Preliminary Explorations.
Technical Report UTEC #82-067, University of Utah, October (Revised), 1982.

[25] Subrahmanyam, P.A.
From Anna+ to Ada: Automating the Synthesis of Ada Package and Task Bodies.
Technical Report Internal Report, University of Utah, March, 1982.

[26] Purushothaman.S, and Subrahmanyam, P.A.
Algebraic Modeling of Self Timed Systems.
Technical Report UTEC #82-066, University of Utah, August, 1982.

[27] Subrahmanyam, P.A.
A Theoretical Basis for the Synthesis and Verification of Systolic Designs.
Technical Report UTEC-82-097, Dept. of Computer Science, University of Utah, June, 1982.

[28] Subrahmanyam, P.A.
Transformational Implementation of Software/Hardware Systems: Global Strategy Guidance.
Submitted for Publication, University of Utah, January, 1982.

[29] Subrahmanyam, P.A.
An Algebraic Basis for VLSI Design.
Draft of a Research Monograph, April 1982. Available from the Department of Computer Science, University of Utah.

[30] J. E. Volder.
The CORDIC Trigonometric Computing Technique.
IRE Transactions on Electronic Computers Volumn Number Unknown:330= 334, September, 1959.

[31] Wile, Dave.
POPART: A Producer of Parsers and Related Tools, System Builder's Manual.
June 1980.
Unpublished, USC/ISI.

ASSASSIN: A CAD System for Self-Timed Control-Unit Design

Tony M. Carter
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

October 1982

Abstract

Many software systems exist for automatically implementing synchronous state-machines. Presented in this paper is a software system — **ASSASSIN** — for the design and automatic layout of self-timed (or speed-independent) control-units as integrated circuit modules. **ASSASSIN** provides for the editing of textual descriptions of control-flow, the functional simulation of speed-independent control-units, and the automatic layout of the implementation as a Path-Programmable Logic (PPL) program. **ASSASSIN** uses a well-known technique (a one-hot state encoding) for implementation of the control-unit. Examples are given illustrating the specification and implementation of simple state-machines. In addition, the design of a state-machine of interest in the University of Utah's Ada-to-Silicon project is carried out. A portion of the Ada[1] code for the "Output Side" of the Inter-Net-Module (INM_OUT), which will eventually be fabricated as part of the Ada-to-Silicon Project, is converted by hand to **ASSASSIN** input format and from there to an integrated circuit layout by **ASSASSIN**, thus illustrating the use of **ASSASSIN** in the context of the Ada-to-Silicon Project.

This work was sponsored in part by the Defense Advanced Research Projects Agency (DARPA) under contract number MDA903-81-C-0414.

1. Introduction

The development of CAD tools for integrated circuit design has exploited a vast body of knowledge about synchronous computing systems. Old and new integrated circuit technologies have been well-suited for implementing synchronous computing systems. The success of these synchronous systems has been prodigious as witnessed by the recent booms in the manufacturing and purchasing of computing systems. Current research in semiconductor devices is rapidly heading toward the ability to construct computing systems which operate orders of magnitude faster and which are far more complex than those currently available. **ASSASSIN** treats part of problem of designing self-timed systems.

With projected room-temperature speeds of logic devices ranging down to tens of picoseconds of delay time [3], it appears that the postulate advanced by Seitz in Chapter 7 of *Introduction to VLSI Systems* [7] will be borne out. The contention is that the current methods of system synchronization (global clocks) will result in unreliable circuits as device speeds increase and as device switching energies decrease.

If Seitz is indeed right, the newer and faster integrated circuit technologies will require computing systems to be implemented using something like "Self-Timed" or "Speed-Independent" logic. In these types of logic, only sequence is of concern. The actual gate and wiring delays will not affect the function, only the absolute speed. It should be noted that any asynchronous device requires that the

[1]Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

surrounding environment to be suitably conditioned so as to tolerate the "un-synchronized" actions of the device.

Much work has been done in the implementation of synchronous structures in integrated circuits. Computing systems can be divided into two main parts: control and data-path. Universities and industry alike have produced many methods for generating synchronous system control, some using the PLA. Work has and is being done in the automatic generation of synchronous data-paths [9]. While there have been some successful efforts to construct self-timed or speed-independent computing systems such as DDM 1 [2] and ILLIAC II [8], there has been very little work done on the implementation of self-timed computing systems in integrated circuits. This may be because there were few integrated circuit implementation strategies which readily lent themselves to the construction of self-timed circuits.

The development of Path-Programmable Logic [1] (PPL), a derivative of the Storage/Logic Array (SLA) [10], has proven to be of great value in the generation of self-timed control in integrated circuits.

ASSASSIN is part of a research effort, being pursued at the University of Utah, to convert Ada programs into integrated circuit implementations. **ASSASSIN** transforms the control portions of Ada programs into their corresponding integrated circuit counterparts. In addition, **ASSASSIN** [1] provides a software tool for the specification, simulation and compilation of self-timed control-units to integrated circuit module layouts. As such, it begins to treat some of the low-level problems of self-timed systems design. It uses PPL as the integrated circuit implementation strategy and a one-hot encoding of the control states [4] as a mapping from the specification to the circuit implementation. It allows an implementation independent specification of control (that is, independent of fabrication technologies and circuit implementation techniques), and provides functional simulation capabilities. Layout generation (analogous to the software compiler code generation) results in self-timed circuits which functionally match the results of simulation. **ASSASSIN** also provides a single, convenient user interface for all of its functions.

2. The Specification of Control: Syntax

The specification of control for a given circuit can result in a labelled, directed graph similar to the one in figure 2-1. There are named nodes which are called states and labelled directed arcs which are called transitions. Associated with states are operations on output variables. These operations may be functions of only the state, or they may be functions of the state and a boolean function of a set of input variables. Transitions are labelled with a boolean function of members of the set of input variables which dictates the condition upon which that transition will take place. Transitions may also have associated operations on outputs (Mealy Machines).

The ability to specify strictly sequential control is certainly essential. Although our current understanding of concurrent processing is very limited, the ability to handle concurrent paths of control may also prove to be useful as our understanding increases. Concurrency (in the context of control) can be interpreted in two ways. The first is where two separate machines operate independently, communicating via some signalling protocol. The second is where a single machine performs some types of concurrent processing by having concurrently executing control paths. The first is handled by having control-units composed of multiple state-machines. In terms of graphs, this implies that one can draw many separate graphs, whose interconnection is implied by output and input variable names. The second is handled by allowing, within a single state-machine, some notion of forking to begin concurrently executing control paths and a notion of joining to terminate concurrently executing control paths. The addition of the concepts of FORK and JOIN to the graph model of control-flow is illustrated in figure 2-2.

Output generation from a control-unit can be either enduring or ephemeral. Enduring outputs

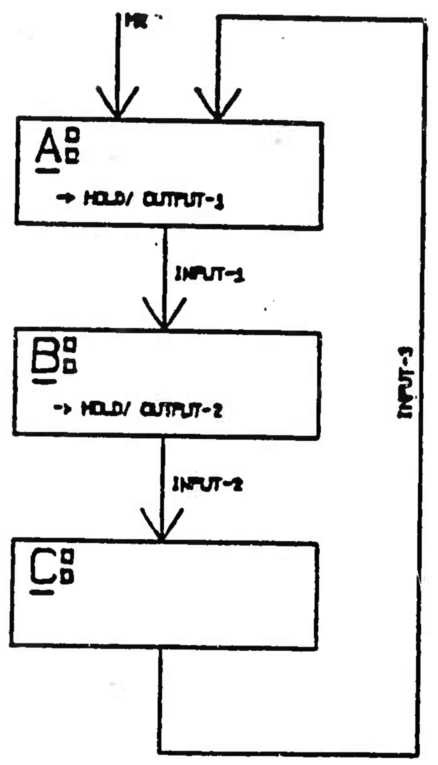

Figure 2–1: A Simple Control–Flow Graph

are latched and operated on by SET and RESET only. When an enduring output is SET it will remain on until a RESET operation is performed. Ephemeral outputs are gated and remain on only while the required condition is met (either residence in a state or execution of a transition). They are operated on by HOLD.

Figure 2–3 contains a control–flow graph which contains all of the features included in the discussion above. States are represented by rectangles with the name of the state indicated in the upper left corner, followed by a colon. Output generation is Indicated by a right–arrow. To the left of the right–arrow will be a boolean expression and to the right the operations to be performed and the names of the outputs which are to be operated on. For example, State B contains three output operations. The first is unconditional (it depends only on the state of the machine) and causes the ephemeral output "O1" to be held true. The second is conditional (the boolean expression is "I3") and causes the enduring output "O3" to be SET. The third is also conditional (the boolean expression is "I4 OR I5") and causes the ephemeral outputs "O2" and "O5" to be held true and the enduring output "O4" to be RESET.

Also required in the specification of control is the concept of an initial state. In the graphs, this is indicated by the arc labelled MasterReset which has no state node at its tail.

In summary, the specification language for control should include the following features:

—the concept of an initial state,

—simple transitions from one state to another (MOVE),

—transitions from one state to many states (FORK),

—transitions from many states to one state (JOIN),

—outputs controlled only by residence in a state or by the execution of a transition,

—outputs controlled by a boolean combination of inputs AND by residence in a state or by the execution of a transition,

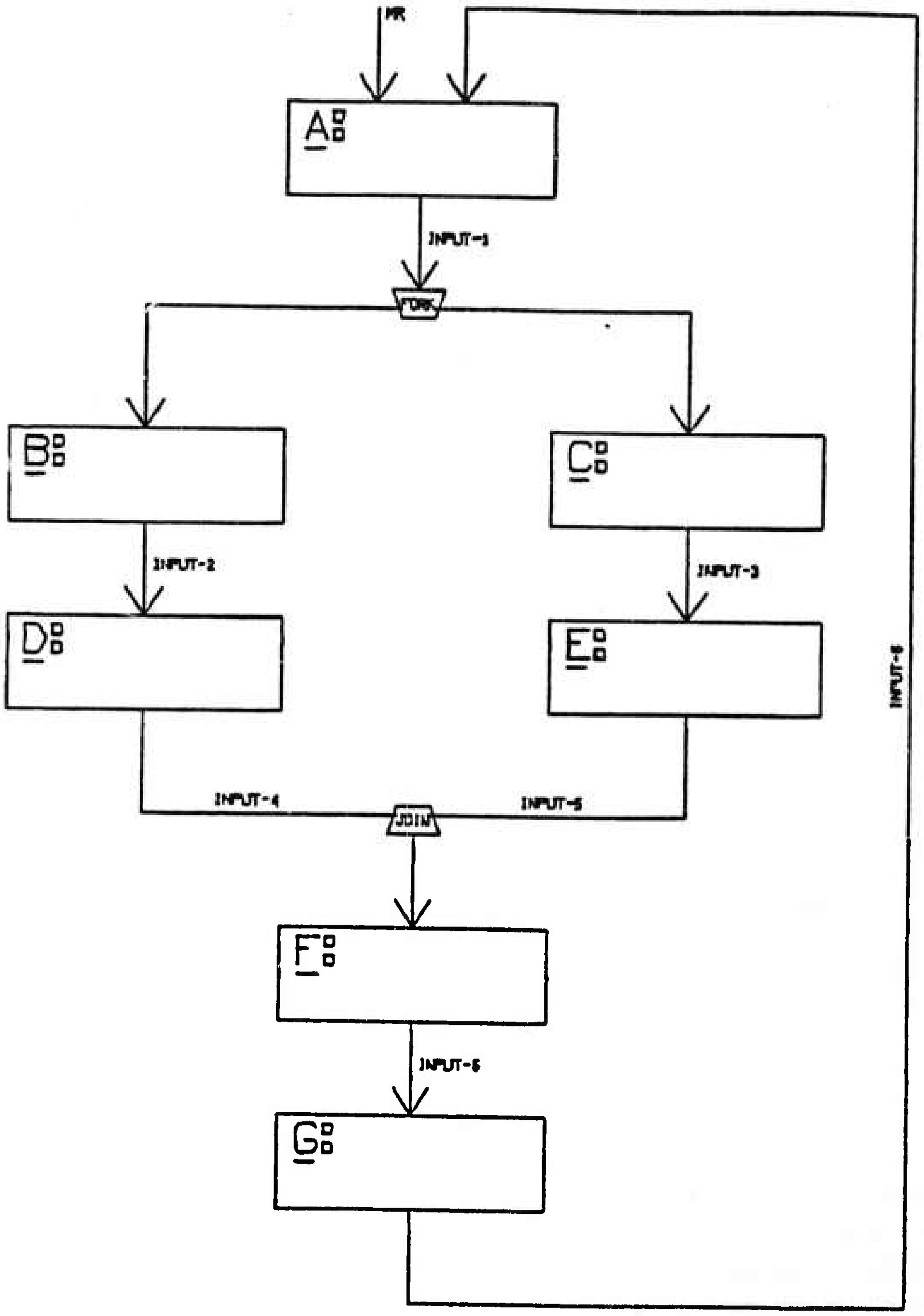

Figure 2–2: A Control–Flow Graph With Concurrency

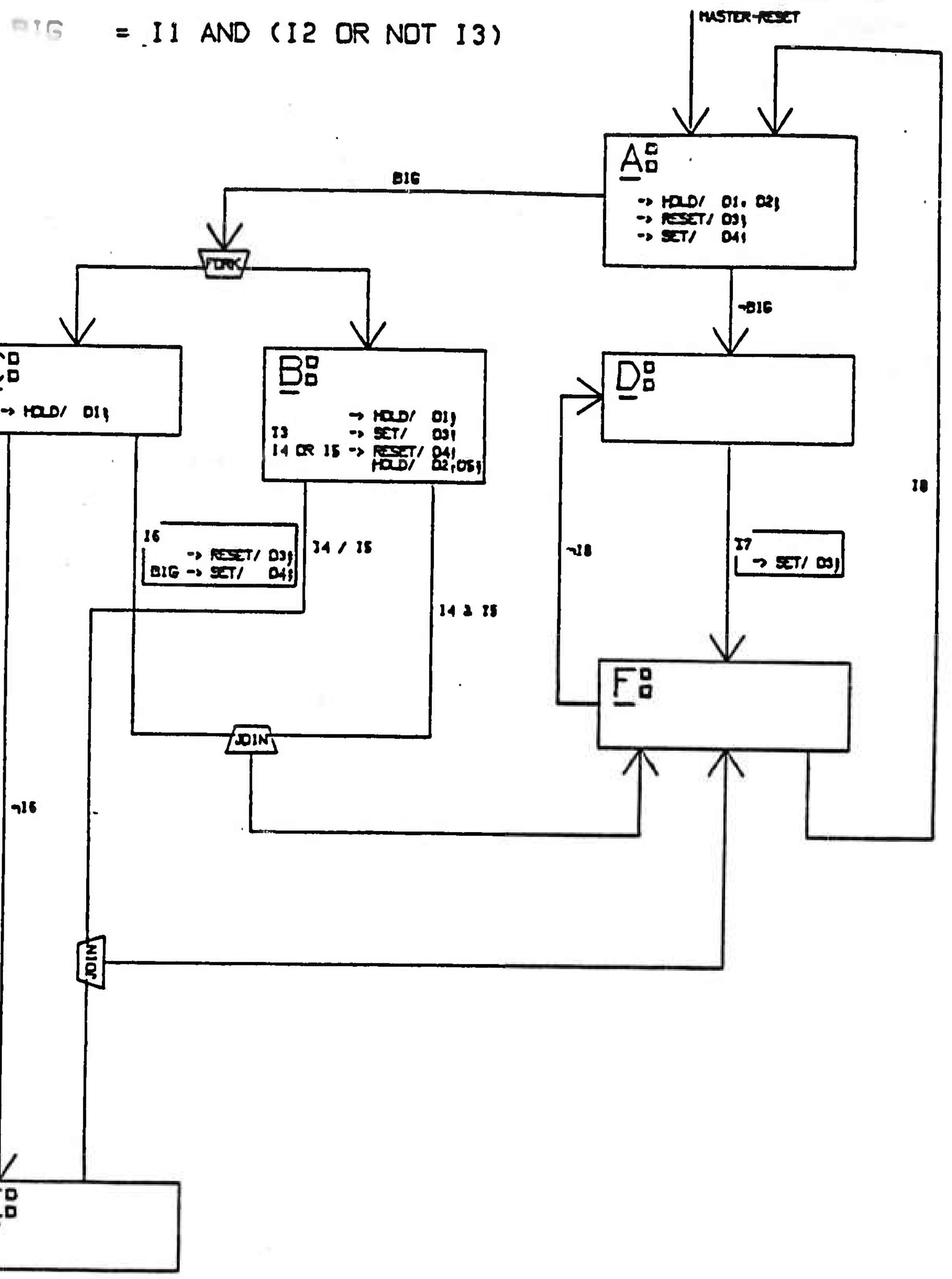

Figure 2-3: A Complex Control-Flow Graph

—arbitrarily complex boolean expressions for conditions (controlling transitions and output generation),

—lambda transitions (where the condition is the tautology TRUE),

—ephemeral outputs,

—enduring outputs,

—multiple and varied transitions from a given state,

—multiple and varied transitions to a given state, and

—multiple state-machine control-units.

The task now is to codify the points listed above, such as in a grammar in BNF. It must allow for all the points listed above while limiting its expressive power to those points. The language must be easily parsed and it is desirable that parser generators be used to generate the code for the parser. Above all, the language should be concise and intelligible to design engineers.

The complete BNF for the language (which is called CUDL) is included in Appendix I. The language has the ability to represent each of the points listed above. There are four types of blocks in the language. The first is the CONTROLUNIT block. This block indicates the name of the overall control-unit and contains STATEMACHINE blocks. It also includes the specification of "global" input expressions which assign boolean expressions to an internal variable which can significantly reduce the size of the code written to describe the control-unit. The names of "global" inputs can be used in the descriptions of transitions and output generation. Figure 2-4 contains the CUDL code describing the machine whose graph is in figure 2-3.

```
controlunit CompileTest9:

inputs: BIG := I1 and (I2 or not I3);

   selftimed statemachine Test9:
      startstate A:
         forkon BIG to B,C;
         moveon NOT BIG to D;
         hold O1,O2;
         reset O3;
         set O4;
      end;

      state B:
         joins C on I4 AND I5 to F;
         joins E on I4 OR I5 to F;
         hold O1;
         if I3 then set O3;
         if I4 OR I5 then begin reset O4; hold O2,O5; end;
      end;

      state C:
         moveon NOT I6 to E;
         joins B on I6 to F doing begin reset O3;
                                        if BIG then set O4; end;
         hold O1;
      end;

      state D:
        moveon I7 to F doing set O3;
      end;

      state E:
        joins B on TRUE to F;
      end;

      state F:
        moveon I8 to A;
        moveon NOT I8 to D;
      end;
   end;
end.
```

Figure 2-4: CUDL Code for the Graph in Figure 2-3

Eventually, given an appropriate display device, a graphical version of this language could be developed and the specification of control could be done in terms of control-flow graphs rather than a textual description of the graph. A project is underway to implement such a front end to **ASSASSIN** on an Apollo DOMAIN computer.

3. The Simulation of Control: Semantics

Given that the syntax of control-unit specification is defined, the designer must also understand the semantics in order to use the system. The semantics of control is directly influenced by the implementation strategy selected. Since the specification of control should allow for concurrency within a given state-machine, a scheme which allows the implementation of such concurrency must be selected. The notion of concurrency eliminates the possibility of completely and uniquely encoding the state variables. The one-hot implementation scheme (completely decoded) allows for easy implementation of concurrency. The following discussion is largely based on the assumption that a one-hot implementation is used.

The specification syntax described in the previous section can be interpreted in three ways. The interpretation depends on the particular mapping strategy being used in the compilation. The three possible types of mapping are synchronous, asynchronous, and self-timed. In order to allow for all three interpretations to be eventually simulated and compiled, the language includes the concept of a state-machine type. The choice of a state-machine level semantic interpretation is made explicit through the use of the keywords: SELFTIMED, ASYNCHRONOUS, and SYNCHRONOUS. In this way, the user can specify various types of control using the same system. Only the SELFTIMED option is currently implemented in **ASSASSIN**.

The simulation of self-timed control can be functional in nature. This functional simulation provides knowledge about the sequential function of the circuit. Since the implementation of the circuit is such that if sequence is correct, function is correct, the user is sure that the circuit will work if the environment in which he places it is conditioned to interact in a self-timed manner with the control-unit.

The simulation of synchronous and asynchronous control really requires the use of a detailed timing simulator. This simulator must be able to make accurate delay calculations based on variable gate delays. In the world of the integrated circuit, these delays may or may not be easily calculated since long wires and heavy loads will significantly alter the operation of any given gate. Thus, the problem of simulation for these types of systems is much more difficult that for the self-timed systems.

To interpret the semantic actions of the control-unit, one must know first the actions to be taken to execute a transition and second how outputs are generated. Transitions are operations that change the internal state of the machine. Although there may be many transitions specified for leaving a given state, it should never be possible to execute two transitions concurrently from the same state. Since the control-unit has no control over the sequence of arrival and the timing of the inputs that trigger transitions, the problem of having two transitions executed simultaneously is inherently a dynamic one and its avoidance requires a detailed knowledge of the environment into which the control-unit is to be placed. If two transitions were executed simultaneously, the result would be a state-machine which would be in two sequential and mutually exclusive states at the same time.

The three interpretations of control have somewhat different views of transitions. The one-hot implementation uses transitions that are essentially handshakes between logically adjacent states. This characteristic can be portrayed by a "token-passing-machine", with provisions made for the controlled splitting and recombination of tokens (FORK and JOIN). In a transition between state A and state B, state A will first set state B and then state B will reset state A. Consider the case (figure

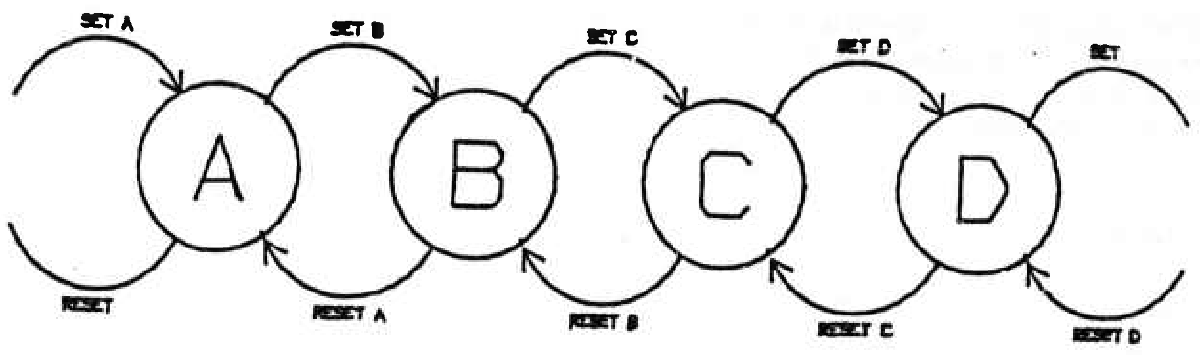

Figure 3–1: Handshaking States

3–1) where a machine contains four sequential states, A, B, C and D. Assume the machine is currently in state B. If a transition is executed, moving from state B to state C, both states B and C will be on during the time it takes state C to reset state B. Now, consider what happens if the transition from state C to state D can occur immediately after state C is set. If state C can set state D and state D can reset state C before state C can reset state B, the machine will be left in a state where both states B and D are on — resulting in a malfunction.

The differences between the three semantic interpretations all center around what to do about this timing problem. In the self–timed approach, it must be guaranteed that such a malfunction cannot occur. In order to ensure this, the state–machine must verify that each transition is complete before allowing the next one. This is done by imposing an additional condition on each transition. It is no longer sufficient just to be in a state for a transition to be possible. In addition, all states which could possibly cause a transition to the current state (its predecessors) must also be off. In the asynchronous approach, it is assumed that gate delays will be well enough behaved that this problem does not arise. This approach is especially naive in the context of the integrated circuit where gate delays may vary nearly an order of magnitude depending on loading. The synchronous approach tries to avoid the problem by recognizing inputs that trigger transitions only at specified times. If the clock period is of the same order as the delays in the faster gates, the problem will not be avoided. Unfortunately, the introduction of the clock necessarily slows the response of the control–unit. Of the three approaches, only the self–timed approach guarantees a control–unit which cannot malfunction due to internal timing problems.

Looking from inside the control–unit, there are two types of outputs. The first is the ephemeral or gated output. It is turned on only while the appropriate condition is met. The second is the enduring or latched output. This type of output is controlled by setting or resetting a latch and therefore its level is maintained even after the appropriate condition has disappeared. It is possible, however, to place a latched output in a metastable condition by trying to set or reset it at the same time, so some care must be taken in working with latched outputs.

The generation of outputs from a control–unit is always conditional upon something. What we term as an unconditional output is an output that depends only on being in a particular state or only on a particular transition being executed. What we term as a conditional output depends not only on state or transition, but also on a boolean combination of input variables.

Unconditional outputs are operated on immediately upon entry into a state or upon the execution of a transition. Also, ephemeral outputs which are unconditionally operated on from a state or transition must be released when the state is left or the transition is completed.

Conditional outputs are operated on when the entire condition becomes true, including entry to a state or execution of the appropriate transition. Again, ephemeral outputs which are conditionally operated on from a state or transition must be released when either the boolean condition is no longer met or the state is left or the transition is completed.

Because of the handshake going on between logically adjacent states, there is a small amount of time when the machine is legally in both states at the same time. This allows for ephemeral outputs to be ORed in a glitch–free manner between logically adjacent states. Enduring outputs controlled by logically adjacent states pose a problem if both a set and reset are attempted at the same time — the output latch will temporarily be placed in a metastable state, possibly adversely affecting the surrounding environment.

In **ASSASSIN**, there is no implicit communication between any two state–machines specified as part of the same control–unit. All such inter–state–machine communication is accomplished by explicit signalling protocols using inputs to and outputs from the state–machines.

4. The Implementation of Control

The actual physical implementation of control depends on two factors: the circuit implementation technique and the control–unit implementation technique. The circuit implementation technique should be picked so as to make the physical realization of the control–unit implementation technique as simple as possible.

The selection of a control–unit implementation technique depends on the set of features to be implemented. Thus, employing FORK and JOIN prohibits using a monolithic, completely encoded control–unit. Including FORK and JOIN in a control–unit implementation technique requires either a very complex strategy for splitting out the concurrent sections of the control into physically (and perhaps logically) separate sections, a partially encoded scheme where the sequential control sections are encoded and the concurrent are not, or a completely decoded machine. The one–hot implementation is a completely decoded scheme in which FORK and JOIN are easily included. The tradeoffs involved in selecting the one–hot strategy are discussed by Hollaar [4].

Basically, the one–hot strategy involves the use of one latch for each state, two gates for each transition, a latch or driver for each output, and one gate for each condition controlling conditional outputs from a given state or transition. For complex machines, the automatic full–custom layout of a one–hot control–unit could be very difficult.

Path–Programmable Logic provides a very regular structure that is particularly well suited for implementing one–hot control–units. In the mapping of control onto PPL using a one–hot encoding, a single latch is used for each state variable. Each transition maps to two PPL row segments, one to set the next state and the other to reset the current state once the next state has been set. In addition, complex boolean conditions on transitions (or on outputs) may require the introduction of temporary gates. In PPL, the AND of several inputs is detected on a single row. The OR is formed on the columns. For this reason, extra PPL columns containing temporary variables must be inserted for forming the OR terms of boolean expressions. Outputs are controlled by using a single PPL row to drive all the unconditional outputs controlled by a state or a transition. Each separate condition for controlling conditional outputs uses a single PPL row.

4.1. The Implementation of Control: Floor Plan

With the basic mapping strategy defined above, we soon see that there are many ways to specify the global organization or floor plan of the control–unit. The one selected for use in **ASSASSIN** was chosen because it appears to be simple. This floor plan (see figure 4–1) has the state latches, temporary variable inverters, and input inverters in a single band across the middle of the control–unit. Output latches and inverters are placed in a band across the top of the control–unit. Inputs arrive from the bottom of the control–unit and outputs are emitted from the top of the control–unit. This stacking of inputs and outputs results in a significantly smaller area and is a direct consequence of using a PPL–like structure for the circuit implementation. State transitions are generated in the bottom half of the control-unit and boolean expressions and outputs are generated between the state latch band and the output band. It is possible to make other area optimizations in the PPL layout of one-hot control-units.

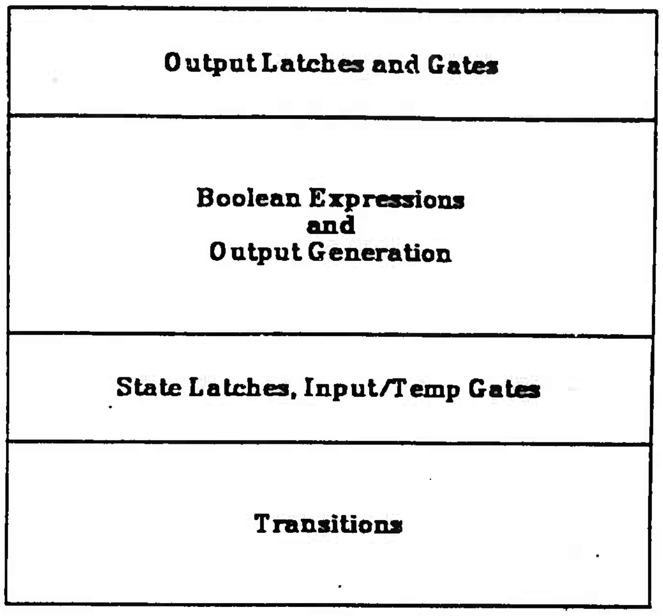
Output Latches and Gates
Boolean Expressions and Output Generation
State Latches, Input/Temp Gates
Transitions

Figure 4-1: Global Organization of ASSASSIN Output

This global organization results in a simple PPL generator that needs no routing tools for constructing the control-unit. All the PPL generator has to know is which cells to place and where to place them — an easy problem when compared with routing.

4.2. The Implementation of Control: Code Generation

We have now almost fully specified the entire system. All that remains is to actually construct algorithms for generating PPL programs that implement the control-unit. The self-timed control-unit requires the use of latches for representing states. These latches must indicate their change in state after the set or reset signal has arrived. The PPL cell designed for this purpose is the four-wire latch. It contains cross-coupled NMOS inverters for the latch with inverting-buffered outputs. Thus, this cell cannot signal its change in state until after the latch has changed state. **ASSASSIN** can currently generate either a CIF description of the control-unit or a file written in Computervision's CADDS2 External Data Base format.

The transitions for a self-timed control-unit require two row segments. The first senses that the machine is in a certain state — say state A, that all possible predecessor states (states which could have caused a transition to state A) have been reset, and that the condition for the transition is met. If all these conditions are met, the latch for the next state is set. If there are outputs controlled by the transition, an inverter is used to appropriately control output generation from the transition. The second row segment detects that the next state has been successfully set and resets state A.

Figure 4-2 illustrates a simple transition between two states. The machine is in state B, having come from state A. State A has been reset. The first row below the state latches performs the "forward" transition, or setting of the next state. The '0' under the latch for state A detects that state A has been reset. The '1' under the latch for state B detects that state B has been set. The '1' under the inverter for input I1 detects that the input condition has been met and the 'S' under the latch for state C will set state C when the transition occurs. The second row performs the "reverse" transition, or the resetting of the previous state. The '1' under the latch for state C detects that state C has been set and the 'R' under the latch for state B will reset state B when the forward transition has been completed. Completing the operations of both these rows constitutes a complete transition.

```
|   | | |   |   |
|   |1|I|   |   |
| A | | | B | C |
|   |1|2|   |   |
|   | | |   |   |
 : : : : : : : :
|Ø---P-1---1---S|
 : : : : : : : :
|-|-|-|-|R---P-1|
```

Figure 4-2: A Simple Self-Timed Transition

Asynchronous transitions are different from self-timed transitions in that they do not sense that predecessor states have been reset. If gate delays are sufficiently non-uniform, a machine constructed in the asynchronous manner would not function properly. Figure 4-3 show the same section of control as in figure 4-2, implemented asynchronously.

```
|   | | |   |   |
|   |I|I|   |   |
| A | | | B | C |
|   |1|2|   |   |
|   | | |   |   |
 : : : : : : : :
|-|-|-|1-P-1---S|
 : : : : : : : :
|-|-|-|-|R---P-1|
```

Figure 4-3: A Simple Asynchronous Transition

Synchronous transitions are implemented the same as asynchronous transitions, with the exception that the state latches are replaced by clocked flip-flops. This is illustrated in figure 4-4.

```
|   | | |   |   |
|   | | |   |   |
|   | | |   |   |
|   | | |   |   |
|   |I|I|   |   |
| A | | | B | C |
-   -1-2-   -   -  Ph12
|   | | |   |   |
-   - - -   -   -  Ph11
|   | | |   |   |
|   | | |   |   |
 : : : : : : : :
|-|-|-|1-P-1---S|
 : : : : : : : :
|-|-|-|-|R---P-1|
```

Figure 4-4: A Simple Synchronous Transition

The following discussion explains the **ASSASSIN** compilation of all the constructs described by Hollaar [4]. Examples are drawn from the sample control-unit whose flow-graph is contained in figure 2-3. The CUDL code for this control-unit is in figure 2-4. The complete PPL program for this example is in figure 4-5. The various constructs being discussed contain portions of this PPL program. Row segments are referred to from left to right in a given row. Row and column numbers are as labeled in the figures.

Figure 4-6 illustrates the compilation of a move transition (from state A to state D). Rows 17 through 19 contain the state latches, input gates and temporary gates. T1 contains "I2 and not I3." T2 contains "I4 or I5." T3 indicates that the JOIN transition from states B and C to state F is currently being taken. T4 indicates that the MOVE transition from state D to state F is being taken. Row 15 is the forward transition from state A to state D. It senses that state A is active by the '1' in column 1, that "BIG" is false by the '0' in columns 2 and 3, and that state F is inactive by the '0' in column 22. State D is made active by the 'S' in column 17 and the row load is the 'P' in column 11. The reverse transition in row 14 simply senses with the '1' in column 17 that state D is active and resets state A with the 'R' in column 0.

Scale-of-two loops pose a particular problem. It is possible to get stuck in both states, with no way to get out. Scale-of-two loops therefore require some sort of mutual exclusion on transitions to avoid this problem. Figure 4-7 illustrates the compilation of a scale-of-two loop. Row 5 contains the forward transition from state D to state F. Note the '0's in columns 0 and 22 which detect the predecessors to state D. The '+' in column 18 is used in generating the outputs associated with this transition by driving T4 when the transition is in progress. The right segment on row 12 resets state D after the forward transition to state F has been finished. Note the '1' in column 19 which senses that input I8 has not yet become false. This gives the required mutual exclusion of input signals in a scale-of-two loop. Row 4 contains the forward transition from state F to state D. The '0' in column 19 detects the false state of input I8 and the other '0's detect the inactivity of the possible predecessors to state F. Row 4 contains the reverse transition associated with the transition from state F to state D. The '0' in column 15 senses that input I7 is currently false.

Figure 4-8 illustrates the FORK transition from state A to states B and C. Row 13 contains the forward FORK transition. It senses the state A is active, that state F is inactive and that input BIG is true (the '1's in columns 2 and 3). It also sets both states B and C. The reverse FORK transition is in the left segment of row 12. It detects that both states B and C have been activated and resets state A.

Figure 4-9 shows the JOIN transition from states B and C to state F. Row 9 implements the forward transition by sensing that the predecessor state (A) is inactive, states B and C are active, inputs I4, I5 and I6 are true, and by setting state F. The '+' in column 14 is used for generating the outputs associated with the JOIN transition from state C. The reverse transition is implemented in row 8 where the activation of state F is detected and states B and C are deactivated (reset).

Figure 4-10 shows the compilation of the input boolean expression BIG – I1 and (I2 or not I3). The leftmost row segments on rows 20 and 21 (I+ –1–PI and I+ –P–0I respectively) compile the subexpression "I2 or not I3." The '+' in column 3 generate the OR of these two rows into T1. I2 is sensed by the '1' in column 4 of row 20 and "not I3" is sensed by the '0' in column 5 of row 21. To sense "BIG", the program must contain '1's in both columns 2 and 3. To sense "not BIG" it must contain '0's in both columns 2 and 3.

Figure 4-11 shows both conditional and unconditional output generation from states and transitions. Row 22 implements the unconditional outputs controlled by state A. The '1' in column 1 senses that state A is active. The '+'s in columns 6 and 13 implement the "HOLD O1,O2;" statement, the 'S' in column 17 implements the "RESET O3" statement and the 'R' in column 10 implements the "SET O4" statement. The 'S' is used to reset a LATCH2 PPL cell and the 'R' is used to set it. Rows 24 and 25 implement the conditional outputs controlled by state B. Row 24 detects the "I4 or I5" condition and HOLDs O5 and O2 and resets O4. Row 25 detects the "I3" condition and sets O3. The last row segment on row 20 (I1–P———SI) implements the unconditional output (O3) controlled by the JOIN transition from states B and C to F. Row 26 implements the "If BIG then set O4" statement from the JOIN transition in state C.

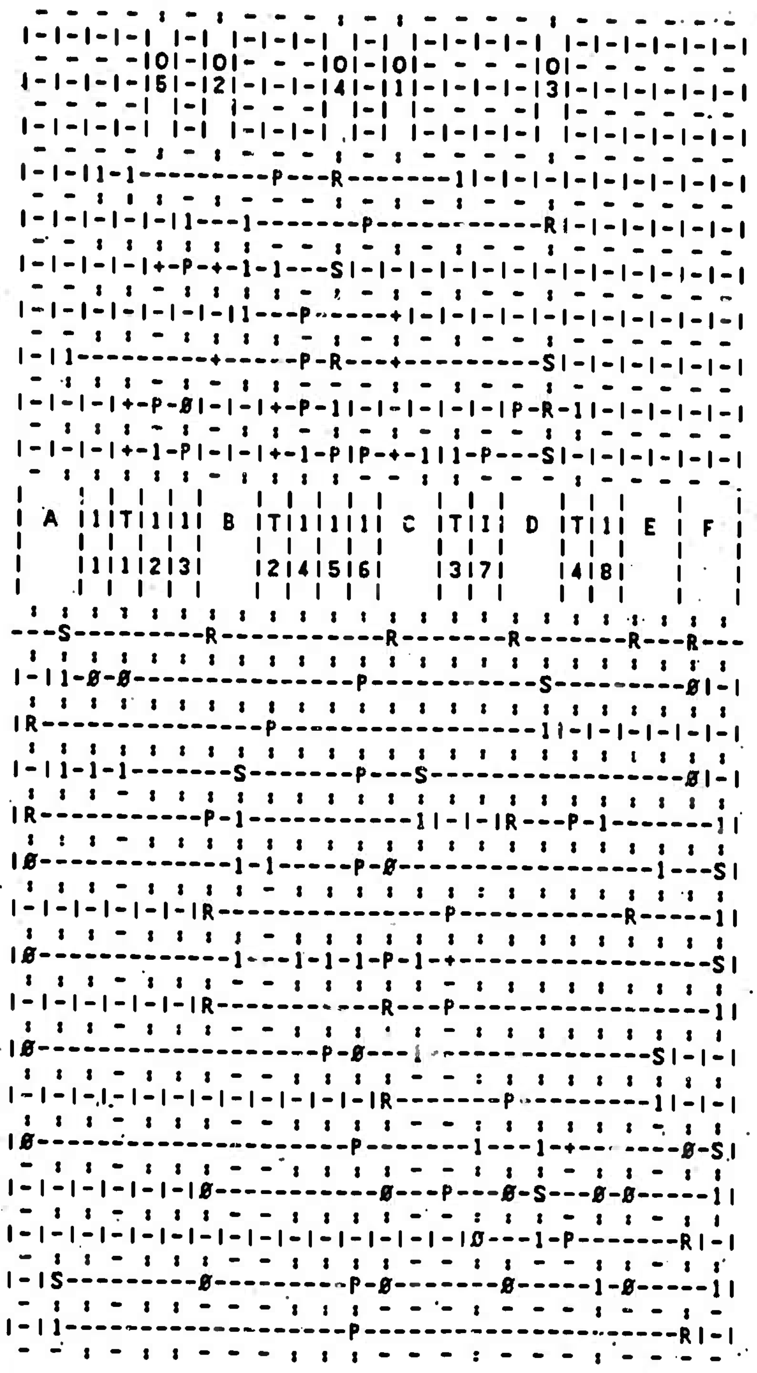

Figure 4-6: Sample PPL Program

```
                           Column Number
                           -------------
                             1 1 1 1 1 1 1 1 1 1 2 2 2 2
         Ø 1 2 3 4 5 6 7 8 9 Ø 1 2 3 4 5 6 7 8 9 Ø 1 2 3

R|  19  |   | | | | |   | | | | |   | | |   | | |   |   |
O|      | A |I|T|I|I| B |T|I|I|I| C |T|I| D |T|I| E | F |
W|  18  |   |1|1|2|3|   |2|4|5|6|   |3|7|   |4|8|   |   |
 |      |   | | | | |   | | | | |   | | |   | | |   |   |
N|  17  |   | | | | |   | | | | |   | | |   | | |   |   |
O|       : : : : : : : : : : : : : : : : : : : : : : : :
.|  15  |-||-Ø-Ø---------------P-----------S---------Ø|-|
         : : : : : : : : : : : : : : : : : : : : : : : :
    14  |R---------------P-----------------1|-|-|-|-|-|-|
```

Figure 4-6: Compilation of the MOVE Transition

```
                           Column Number
                           -------------
                             1 1 1 1 1 1 1 1 1 1 2 2 2 2
         Ø 1 2 3 4 5 6 7 8 9 Ø 1 2 3 4 5 6 7 8 9 Ø 1 2 3

R|  19  |   | | | | |   | | | | |   | | |   | | |   |   |
O|      | A |I|T|I|I| B |T|I|I|I| C |T|I| D |T|I| E | F |
W|  18  |   |1|1|2|3|   |2|4|5|6|   |3|7|   |4|8|   |   |
 |      |   | | | | |   | | | | |   | | |   | | |   |   |
N|  17  |   | | | | |   | | | | |   | | |   | | |   |   |
O|       : : : : : : : : : : : : : : : : : : : : : : : :
.|  12  |R-----------P-1-----------1|-|-|R---P-1-------1|
         : : : : : : : : : : : : : : : : : : : : : : : :
     5  |Ø---------------------P-------1---1-+-------Ø-S|
         - : : - : : : - - : : : : - - : : : - : : - : :
     4  |-|-|-|-|-|-|Ø-----------Ø---P---Ø-S---Ø-Ø-----1|
         - : : - : : : - - : : : : - - : : : - : : - : :
     3  |-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|Ø---1-P-------R|-|
```

Figure 4-7: Compilation of the Scale-of-Two Loop

```
                           Column Number
                           -------------
                             1 1 1 1 1 1 1 1 1 1 2 2 2 2
         Ø 1 2 3 4 5 6 7 8 9 Ø 1 2 3 4 5 6 7 8 9 Ø 1 2 3

R|  19  |   | | | | |   | | | | |   | | |   | | |   |   |
O|      | A |I|T|I|I| B |T|I|I|I| C |T|I| D |T|I| E | F |
W|  18  |   |1|1|2|3|   |2|4|5|6|   |3|7|   |4|8|   |   |
 |      |   | | | | |   | | | | |   | | |   | | |   |   |
N|  17  |   | | | | |   | | | | |   | | |   | | |   |   |
O|       : : : : : : : : : : : : : : : : : : : : : : : :
.|  13  |-|1-1-1-------S-------P---S-----------------Ø|-|
         : : : : : : : : : : : : : : : : : : : : : : : :
    12  |R-----------P-1-----------1|-|-|R---P-1-------1|
```

Figure 4-8: Compilation of the FORK Transition

5. The Assassination of a Control Unit

This section illustrates the complete design of a non-trivial state-machine. The control-unit to be designed comes from the Ada-to-Silicon Project underway at the University of Utah. This project has as one of its objectives the automatic transformation of Ada programs into hardware implementations using integrated circuits [5]. The Ada-to-Silicon project is using the Internet Protocol (see

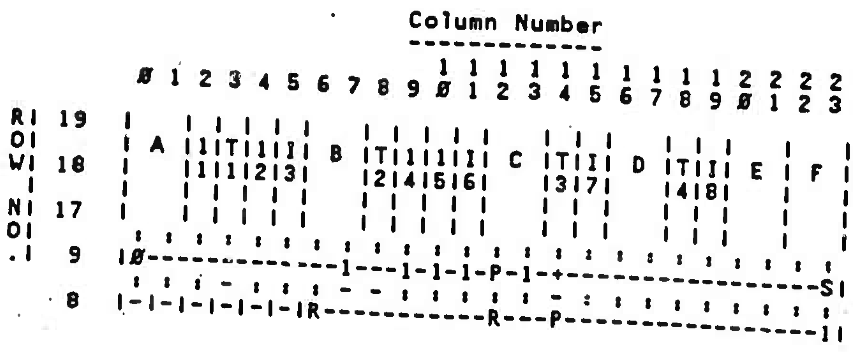

Figure 4-9: Compilation of the JOIN Transition

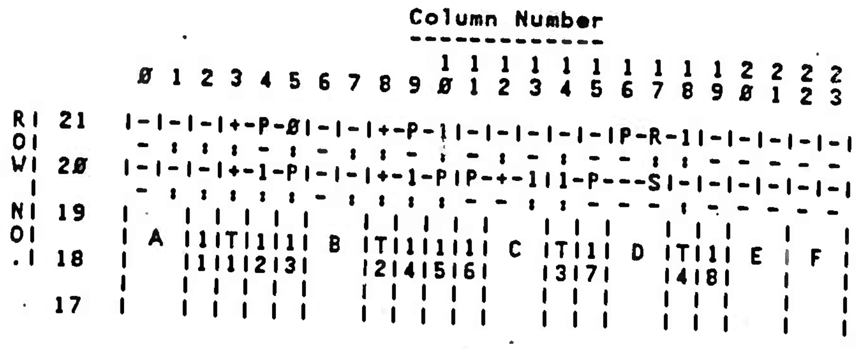

Figure 4-10: Compilation of Boolean Expressions – BIG

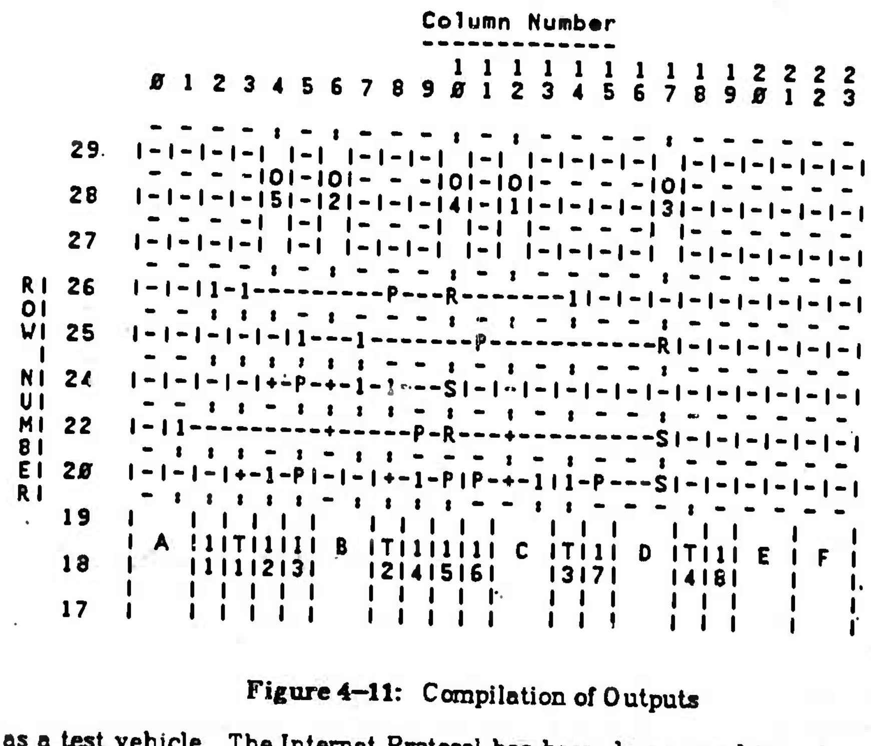

Figure 4-11: Compilation of Outputs

Postel [6]) as a test vehicle. The Internet Protocol has been decomposed into three communicating hardware (and software) submodules [5]. Figure 5-1 illustrates this division. The protocol consists of N INM_IN submodules, each of which receives transmitted data and assembles datagrams from a single local area network, N INM_OUT submodules, each of which appropriately fragments and transmits datagrams to a single local area network, and a single INM_SRV submodule that interfaces the N INM_OUT and N INM_IN submodules to one or more host computers. The complete Ada code describing the INM_OUT submodule has been written and compiled and willis presented in a forthcoming report.

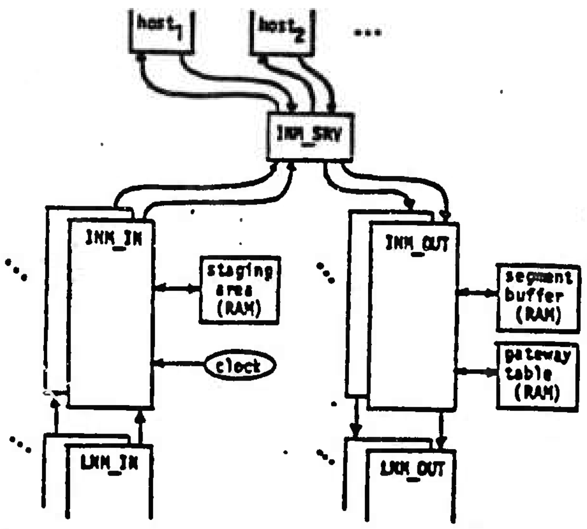

Figure 5-1: Internet Protocol Hardware Submodules

The INM_OUT submodule of the Internet Protocol has been selected as the initial test case. Preliminary Ada code in the form of a complete task has been written and compiled. INM_OUT consists of three separate tasks, Main, Read_Init_Parameters and Translate_TOS_Table. Of these, the hardware architectural design has been completed for the Read_Init_Parameters task. Read_Init_Parameters deals with the initialization parameters of INM_OUT and loads various registers with data related to the transmission of datagrams through a local area network. Illustrated in figure 5-3 is a block diagram of the hardware implementation of this task. Professor Al Davis performed the mapping of the initial version of Ada code into a block diagram. Several modifications have been made since that time. The block marked "Read_Init_Pars - FSM" is the control-unit derived from the Ada code for the Read_Init_Parameters task. Figure 5-2 contains the Ada code for a section of Read_Init_Parameters. The complete code is found in Appendix .

Figure 5-4 contains the control flow-graph for the Read_Init_Parameters task as extracted from the Ada program. It should be noted that this particular flow-graph does not use the FORK and JOIN transitions available in CUDL. Indeed, FORK and JOIN will probably not be used in implementing tasking, but may be used for more fine grained parallelism based on data independency. Ada accept statements are translated into request-acknowledge handshakes with the appropriate module. These are indicated by the name of the accept (GO or SRV) concatenated with ".REQ" and ".ACK". State RIP0 is the initial state of the machine and sends initialization signals to several of the datapath modules in the environment of Read_Init_Parameters. Of particular interest, the signal INITNUM.REG.LOD is held during this state. This signal indicates to the register holding the initialization number to watch the associated three-wire bus and assume its value at all times. When this signal is dropped (in state RIP1), this register latches the value on the bus. The first accept statement ("accept GO(...) ") is begun with the transition from state RIP0 to state RIP1.

```
begin
  loop
    accept Go(  -- Get OP Code from Main (size of Memory Address)
         init_num_formal:            bit8;
         response:              out out_response)
      do
        response := sent_ok;                               -- Also assns init_ok.
        for index in 1 .. init_num_formal
        loop
          accept Srv_req( -- Process Memory Address
               server_command_datum:      srv_command;
               response_to_server:    out out_response)
          do
            Memory_request( -- Put chunk out to the Memory module.
                 request_type_formal      => load_address,
                 chunk_of_address_formal  => server_command_datum,
                 octet_formal             => dont_care_octet);
          end Srv_req;
        end loop;

        -- Get the 6 individual initialization parameters (contained in the
        -- next 8 octets received) from the Memory Module.
        for index in 0 .. 7
        loop

          Memory_request(
               request_type_formal      => receive_datum_octet,
               chunk_of_address_formal  => dont_care_X_datum,
               octet_formal             => octet_register);

          case index is
            when 1 => lns_max_packet.lo          := octet_register; -- 8 bits
            when 2 => lns_max_packet.hi          := octet_register; -- 8 bits
            when 3 => lns_address_length         := octet_register; -- 8 bits
            when 4 => lns_time_out.lo            := octet_register; -- 8 bits
            when 5 => lns_time_out.hi            := octet_register; -- 8 bits
            when 6 => ack_type                   := octet_register; -- 1 bit
            when 7 => local_net_type_of_service_table_row_size
                                                := octet_register; -- 3 bits
            when 8 => number_of_local_net_types_of_service
                                                := octet_register; -- 3 bits
          end case;
        end loop;

        -- Read in type-of-service translation table.

        declare
          row_number: integer range 0 .. number_of_local_net_types_of service;
          col_number: integer range 0 .. local_net_type_of_service_row_size;

          index:      integer range 0 .. number_of_local_net_types_of service
                                         * local_net_type_of_service_row_size
                                         := 0;
        begin
          row_number := 0;
          loop                        -- Outer loop reads all rows of TOS table.
            col_number := 0;
            loop                      -- Inner loop reads in one row of TOS table.
              Memory_request(
                   request_type_formal      => receive_datum_octet,
                   chunk_of_address_formal  => dont_care_X_datum,
                   octet_formal             => tos_table(index));
              col_number := col_number + 1;
              exit when col_number = local_net_type_of_service_row_size;
              index := index + 1;
              if  index > max_tos_table_size  then
                response := bad_srv_command;
                return;               -- Exit the current accept statement.
              end if;
            end loop;                 -- End inner loop.

            row_number := row_number + 1;
            exit when row_number = number_of_types_of_services;
          end loop;                   -- End outer loop.
        end;                          -- End declare block.
      end Go;                         -- End of init processing.
    end loop;                         -- End of outer-most (infinite) loop.
end Read_Init_Parameters;
```

Figure 5-2: ADA Code for Read_ Init_ Parameters

Note that the condition for the transition includes, in addition to GO.REQ, INITNUM.REG.DON and INITNUM.CTR.DON. The machine cannot proceed until it is sure that the initialization number register contains the correct value and the associated counter has been reset. In state RIP1, the machine begins the second accept loop. When the SRV.REQ signal arrives, a transition is made to

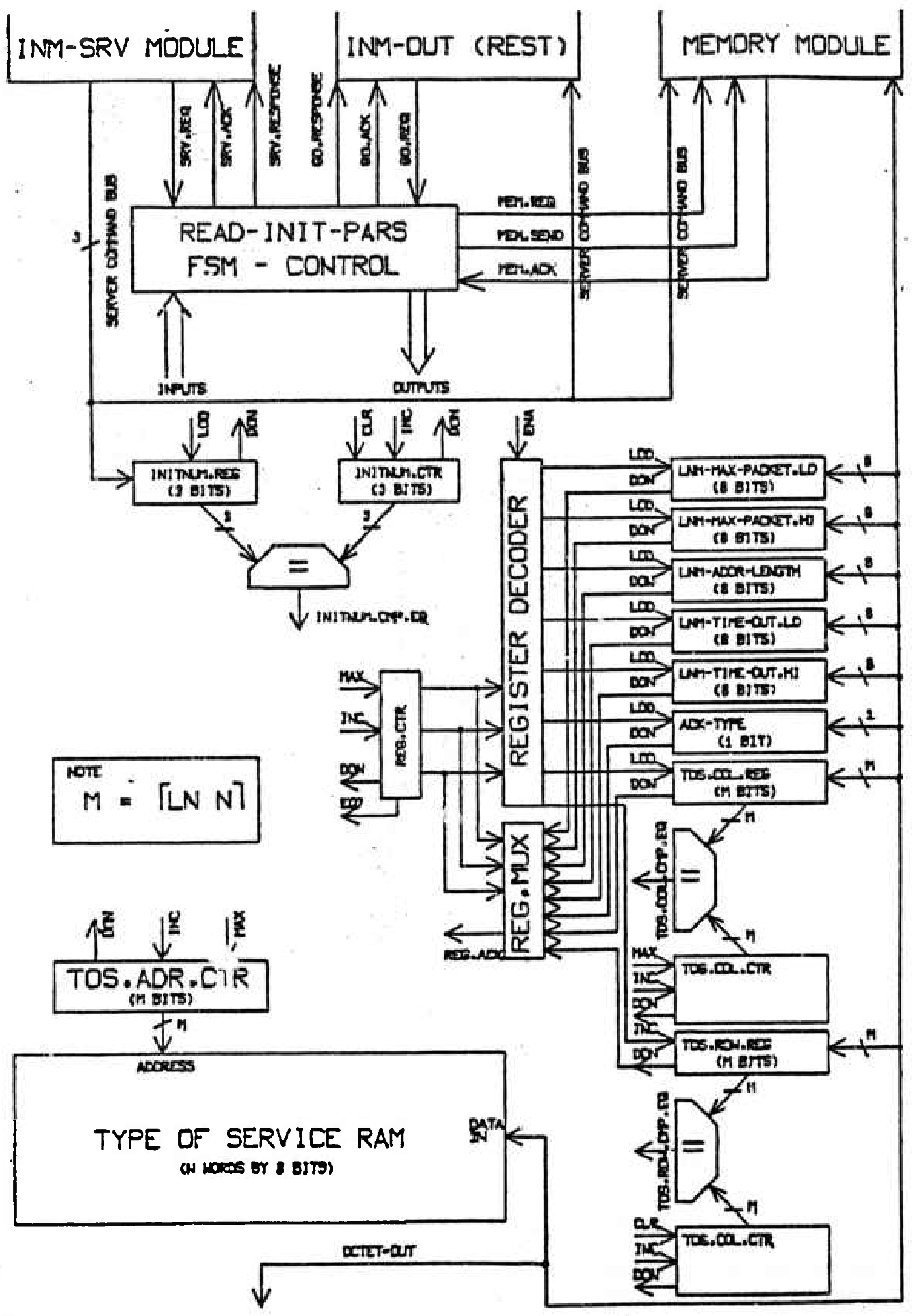

Figure 5-3: Block Diagram of Read_ Init_ Parameters

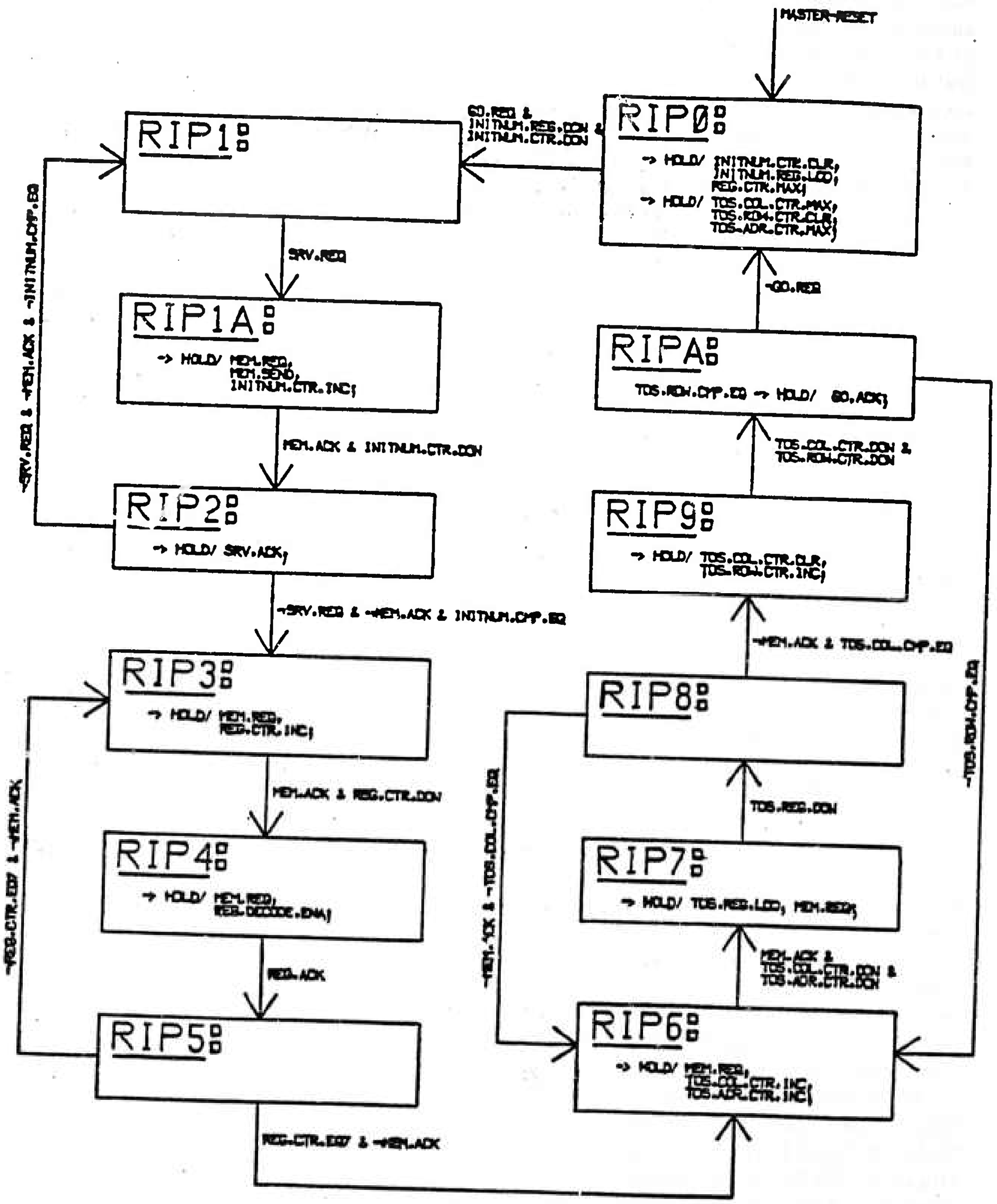
MASTER-RESET
RIP0:
-> HOLD/ INITNUM.CTR.CLR, INITNUM.REG.LOD; REG.CTR.MAX;
-> HOLD/ TOS.COL.CTR.MAX, TOS.ROW.CTR.CLR; TOS.ADR.CTR.MAX;
GO.REQ & INITNUM.REG.DON & INITNUM.CTR.DON
RIP1:
SRV.REQ
RIP1A:
-> HOLD/ MEM.REQ, MEM.SEND, INITNUM.CTR.INC;
MEM.ACK & INITNUM.CTR.DON
RIP2:
-> HOLD/ SRV.ACK;
~SRV.REQ & ~MEM.ACK & ~INITNUM.CMP.EQ
~SRV.REQ & ~MEM.ACK & INITNUM.CMP.EQ
RIP3:
-> HOLD/ MEM.REQ, REG.CTR.INC;
MEM.ACK & REG.CTR.DON
RIP4:
-> HOLD/ MEM.REQ, REG.DECODE.ENA;
REG.ACK
RIP5:
~REG.CTR.EQ7 & ~MEM.ACK
REG.CTR.EQ7 & ~MEM.ACK
RIP6:
-> HOLD/ MEM.REQ, TOS.COL.CTR.INC, TOS.ADR.CTR.INC;
MEM.ACK & TOS.COL.CTR.DON & TOS.ADR.CTR.DON
RIP7:
-> HOLD/ TOS.REG.LOD, MEM.REQ;
TOS.REG.DON
RIP8:
~MEM.ACK & ~TOS.COL.CMP.EQ
~MEM.ACK & TOS.COL.CMP.EQ
RIP9:
-> HOLD/ TOS.COL.CTR.CLR, TOS.ROW.CTR.INC;
TOS.COL.CTR.DON & TOS.ROW.CTR.DON
RIPA:
TOS.ROW.CMP.EQ -> HOLD/ GO.ACK;
~TOS.ROW.CMP.EQ
~GO.REQ

Figure 5-4: Control Flow-Graph for Read_ Init_ Parameters

state RIP2, where the counter is incremented (indicating that another byte of address is to be transmitted to the memory module), and a request-acknowledge handshake is performed between READ_INIT_PARS and the memory module. The signal MEM.SEND indicates to the memory that it is to receive data. When the counter has been incremented (INITNUM.CTR.DON) and an acknowledgement from the memory (MEM.ACK) have been received, a transition is made to state RIP2. State RIP2 terminates the handshake with the INM_SRV module by asserting the signal SRV.ACK. Once both SRV.REQ and MEM.ACK have been lowered, the output of the comparator between the initialization counter and register is examined. One of the two transitions from state RIP2 is executed based on the value of INITNUM.CMP.EQ. If INITNUM.CMP.EQ is on, the initialization loop is terminated. If it is off, the initialization loop is continued.

The memory module now has the complete address of the parameter block which needs to be transmitted to INM_OUT. State RIP3 begins an interaction between the memory module and Read_Init_Parameters that loads a set of registers appropriately. The handshake with the memory module is begun by holding MEM.REQ. At the same time, the register counter (which was initialized to 7) is incremented (and is now 0). When an acknowledgement is received from the memory (MEM.ACK), and the register counter is finished counting up by one, a transition is made to state RIP4 where the signal REG.DECODE.ENA signals the appropriate latch to gate in the value from the memory bus. MEM.REQ is left on here so that the valid data on the memory bus does not disappear before it can be latched. When the appropriate register signals that it has the data loaded (REG.ACK), a transition is made to state RIP5. When the memory acknowledges the termination of a transmission cycle (not MEM.ACK), a comparator with the register counter is made to see if all required registers have been loaded (REG.CTR.EQ7). If not, the loop is repeated, incrementing the register counter each time. If so, a transition is made to state RIP6 and the processing of the Type-of-Service (TOS) table is performed.

The type-of-service table is to be a linear array of registers (or ram cells), indexed by row and column. Initially this indexing was done via a multiplication (in the Ada code). It was replaced with a doubly nested loop to make the hardware implementation easier and more straightforward. In state RIP6, the type-of-service column counter and type-of-service address counter are incremented. They were initialized to their maximum value in state RIP0. At the same time, a handshake with the memory module is begun (by raising MEM.REQ). When the memory has placed the data on the line and replied by using MEM.ACK, and when the two counters, TOS.COL.CTR and TOS.ADR.CTR have been incremented, a transition is made to state RIP7. Here the TOS table is signalled to load the value from the memory bus (TOS.REG.LOD). MEM.REQ is held high so that the data on the memory bus remains valid. When the data is in the TOS table, TOS.REG.DON is asserted and the next state becomes RIP8. This state terminates the handshake with the memory module. When the acknowledgement from the memory arrives, if all columns in the current TOS table entry have been processed, a transition is made to state RIP9 to proceed to the next TOS table entry. If more columns in the entry need to be processed, the TOS.COL.CMP.EQ signal will be false and the transition from state RIP8 to state RIP6 will be taken.

In state RIP9, the column counter (TOS.COL.CTR) is cleared and the row counter (TOS.ROW.CTR) is incremented. When these two operations are complete, the next state becomes RIPA where a check is performed to see of the entire TOS table has been loaded. If it has not, TOS.ROW.CMP.EQ will be false and the a transition occurs from state RIPA to state RIP6. If TOS.ROW.CMP.EQ is true, the output GO.ACK is asserted, terminating the "Accept GO (...)" statement. When GO.REQ is lowered, the next state becomes RIP0 to begin over again when necessary. Figure 5-6 contains the CUDL code for the Read_Init_Parameters state machine.

The CUDL code in figure 5-6 was run through **ASSASSIN**. The code was simulated to verify that it matched the flow-graph; the associated PPL program was then generated through compilation of the CUDL code. Figure 5-6 contains a plot of the PPL program for the Read_Init_Parameters control.

```
ControlUnit ReadInitParms:

 StateMachine RIP:

   StartState RIP0:
     moveon GO_Req and (InitNum_REG_DON and InitNum_CTR_DON) to RIP1;
     hold InitNum_CTR_CLR, InitNum_REG_LOD, REG_CTR_MAX;
     hold TOS_Col_CTR_MAX, TOS_Row_CTR_CLR, TOS_ADR_CTR_MAX;
   end;

   state RIP1:
     moveon SRV_Req to RIP1A;
   end;

   state RIP1A:
     moveon MEM_Ack and InitNum_CTR_DON to RIP2;
     hold MEM_Req, MEM_Send, InitNum_CTR_INC;
     set GO_Response;
   end;

   state RIP2:
     moveon not SRV_Req and (not MEM_Ack and     InitNum_CMP_EQ) to RIP3;
     moveon not SRV_Req and (not MEM_Ack and not InitNum_CMP_EQ) to RIP1;
     hold SRV_Ack;
   end;

   state RIP3:
     moveon MEM_Ack and Reg_CTR_DON to RIP4;
     hold MEM_Req, Reg_CTR_INC;
   end;

   state RIP4:
     moveon Reg_ACK to Rip5;
     hold MEM_Req, Reg_Decode_ENA;
   end;

   state RIP5:
     moveon     Reg_CTR_EQ7 and not MEM_Ack to RIP6;
     moveon not Reg_CTR_EQ7 and not MEM_Ack to RIP3;
   end;

   state RIP6:
     moveon MEM_Ack and (TOS_Col_CTR_DON and TOS_Adr_CTR_DON) to RIP7;
     hold MEM_Req, TOS_Col_CTR_INC, TOS_Adr_CTR_INC;
   end;

   state RIP7:
     moveon TOS_Reg_DON to RIP8;
     hold TOS_Reg_LOD, MEM_Req;
   end;

   state RIP8:
     moveon not MEM_Ack and     TOS_Col_CMP_EQ to RIP9;
     moveon not MEM_Ack and not TOS_Col_CMP_EQ to RIP6;
   end;

   state RIP9:
     moveon TOS_Col_CTR_DON and TOS_Row_CTR_DON to RIPA;
     hold TOS_Col_CTR_MAX, TOS_Row_CTR_INC;
   end;

   state RIPA:
     moveon not TOS_Row_CMP_EQ to RIP6;
     moveon not GO_Req to RIP0;
     if TOS_Row_CMP_EQ then hold GO_Ack;
   end;
 end;
end.
```

Figure 5-6: CUDL Code for Read_ Init_ Parameters Control

Figure 5-7 shows the composite layout.

The compilation of the control unit took approximately 2 minutes of DEC-System 20 CPU time. The resulting circuit is 2028 microns by 1050 microns (39 PPL columns by 30 PPL rows using 6-micron geometry). The datapath related to the Read_ Init_ Parameters task cannot be layed out until the relationship of some of the registers, which represent global variables (with respect to

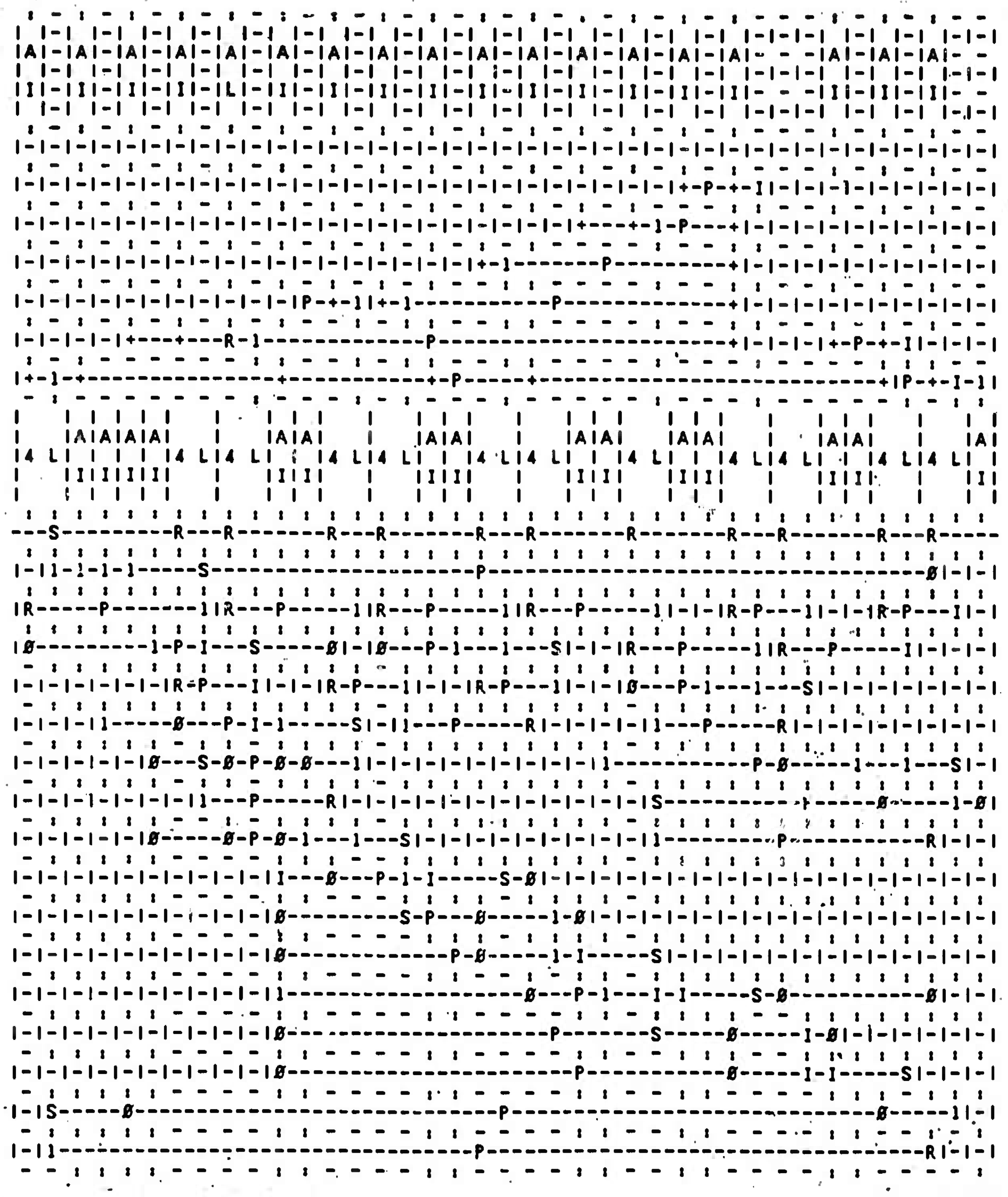

Figure 5-6: PPL Program for Read_ Init_ Parameters Control

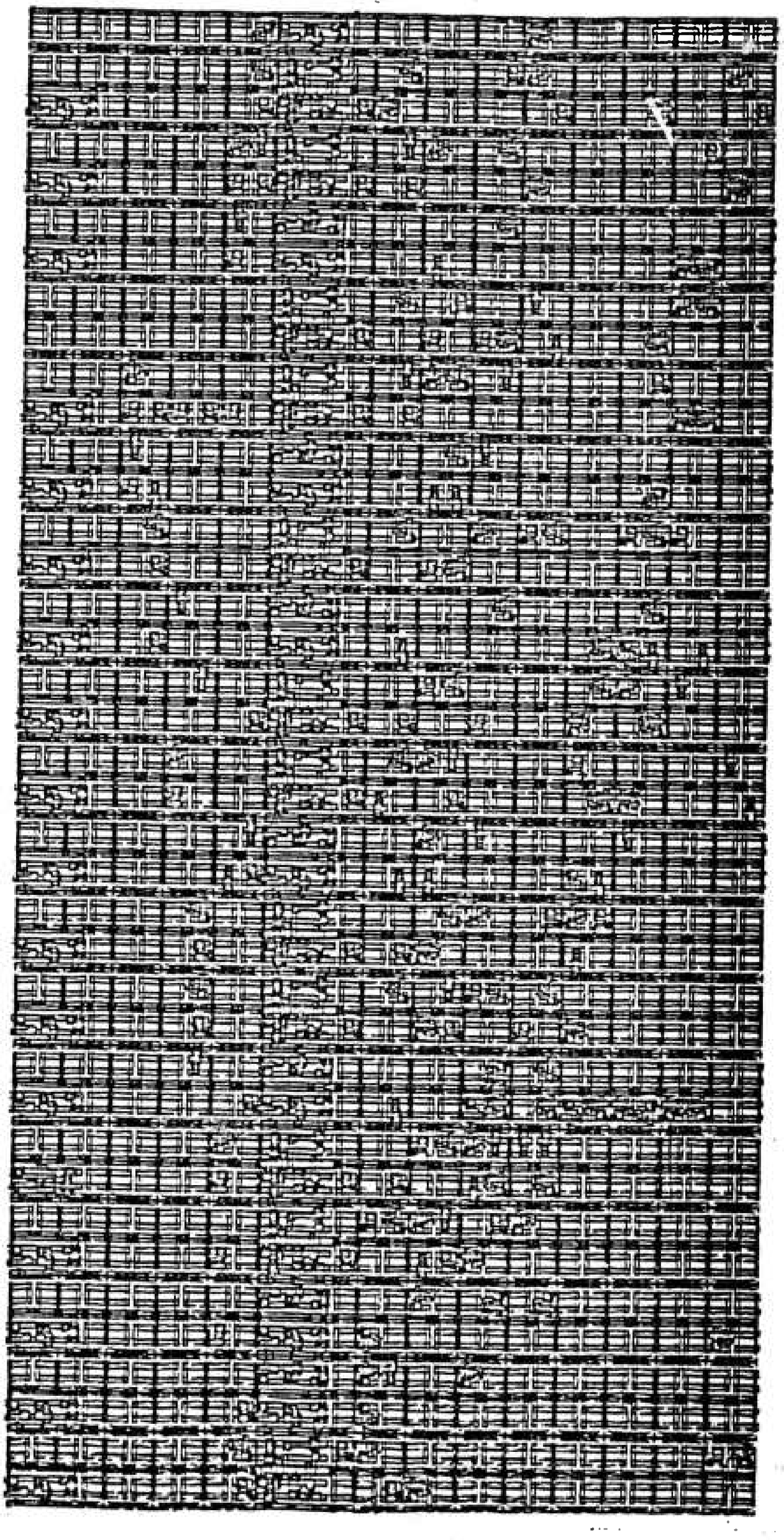
Figure 5–7: Composite Layout (NMOS) for Read_ Init_ Parameters Control

Read_ Init_ Parameters), with other associated control and datapath elements has been established.

6. Conclusions

ASSASSIN demonstrates several significant points.

1. Control can be specified at an abstract level and then automatically and easily implemented as an integrated circuit module. It is possible to map control specified at even higher levels of abstraction to something ASSASSIN understands, thereby enabling us to make progress toward a true silicon compiler. Such work is reported in [11].
2. Self-timed (or asynchronous) control-units with concurrency can be easily implemented. ASSASSIN shows that the control for self-timed machines can be designed with relative ease.
3. The successful use of Path-Programmable Logic in ASSASSIN shows that PPL has great value as a circuit implementation technique, at least for this type of control-unit. This also shows that PPL is indeed amenable to the development of sophisticated CAD tools that use it as the underlying circuit implementation technique.
4. The mapping of Ada's rich set of control constructs is very straightforward as illustrated by the generation of the control for the Read_ Init_ Parameters task. ASSASSIN represents a step forward in the design of integrated circuits by allowing high level descriptions of integrated circuit modules to be automatically compiled to a layout.

7. Acknowledgements

I would like to acknowledge the help of Dr. Lee Hollaar without whose help and encouragement this work would not have been done. The work of Dr. Kent Smith on the development of PPL is greatly appreciated. Acknowledgements are also due to Dr. Elliott Organick, Dr. Alan Davis, Dr. Gary Lindstrom, and Dr. Alan Hayes for work on development of the example of the transformation of Ada to a hardware module.

I. The Syntax for ASSASSIN

The following is a BNF description of CUDL – the Control Unit Description Language. The following are to be used in understanding this description:

```
<>    - a non-terminal symbol
{}    - 0 or more repetitions
:=    - is defined as
|     - OR

language terminals are indicated by uppercase
----------------------------------------------------------------
```

```
<control-unit>         := CONTROLUNIT <identifier> :
                          {<input-descriptor>} <sm-list> END .
<identifier>           := <letter> <id-tail>
<id-tail>              := <letter> <id-tail> | <digit> <id-tail> |
                          <letter> | <digit>
<input-descriptor>     := INPUTS: <input-reduction-list>
<input-reduction-list>:= <reduction-statement>
                                <input-reduction-list> |
                          <reduction-statement>
<reduction-statement> := <identifier> := <condition> ;
<condition>            := <term> OR <condition> | <term>
<term>                 := <primary> | <primary> AND <term>
<primary>              := <identifier> | (<condition>) |
                          NOT <primary> | TRUE | FALSE
<sm-list>              := <sm-descriptor> |
                          <sm-descriptor> <sm-list>
<sm-descriptor>        := <sm-type> STATEMACHINE <identifier> :
                          <state-list> END ;
<sm-type>              := SELFTIMED | ASYNCHRONOUS | SYNCHRONOUS
<state-list>           := <state-descriptor> |
                          <state-descriptor> <state-list>
<state-descriptor>     := STARTSTATE <state-name> :
                                <statement-list> END ; |
                          STATE <state-name> :
                                <statement-list> END ;
<state-name-list>      := <state-name> , <state-name-list> |
                          <state-name>
<state-name>           := <identifier>
<etatement-liet>       := <statement> ; <statement-list> |
                          <statement>
<statement>            := <transition-statement> |
                          <action-statement>
```

```
<transition-statement>:=  <transition-op> <transition>

<transition-op>       :=  MOVEDN | FORKDN |
                          JOINS <state-name-list> ON

<transition>          :=  <condition> TD <state-nams-list> ; |
                          <condition> TD <state-name-list>
                               DOING <action-statsment-list> ;

<action-statement-list>:=  <action-statement> |
                          BEGIN {<action-statement> ;} END

<action-statement>    :=  <action-op> <output-list> |
                          <if-action-statement>

<action-op>           :=  HOLD | SET | RESET

<output-list>         :=  <output-name> , <output-list> |
                          <output-nams>

<output-name>         :=  <identifier>

<if-action-statement> :=  IF <condition> THEN
                               <action-statement-list>;
```

II. Ada Code for the Read_ Init_ Parameters Task of the INM_ OUT Submodule

```
separate (Inm_Out_Module)

task body Read_Init_Parameters is

  -- Accessed globals:
  ------------------
  -- number_of_local_net_types_of_service:          octet_type
  -- local_net_type_of_service_table_row_size:      octet_type
  -- tos_table:                                     octet_buffer_type

  -- Local variable declaration:
  ---------------------------
  -- The following variable is commented out. It appeared only in the
  -- "high-level" used to read in the TOS table.  See below.
  --  number_of_tos_table_octets: integer range 2 .. max_tos_table_size - 1;
  octet_register:                  octet_type;

begin
  loop
    accept Go(
        init_num_formal:        bit3;
        response:           out out_response)
      do
        response := sent_ok;                        -- Also means init_ok.

        -- Get from the server all of the addr_chunks needed to form the
        -- base address in memory that holds the initialization parameters
        -- and sends these chunks to the Memory module.
        for index in 1 .. init_num_formal
        loop
          accept Srv_req(                          -- Get next address
                                                   -- chunk from the
                                                   -- Server Module.
              server_command_datum:     srv_command;
              response_to_server:   out out_response)
          do
            Memory_request(                        -- Put chunk out to
                                                   -- the Memory module.
              request_type_formal       => load_address,
              chunk_of_address_formal   => server_command_datum,
              octet_formal              => dont_care_octet);
          end Srv_req;
        end loop;

        -- Get the 6 individual initialization parameters (contained in the
        -- next 8 octets received) from the Memory Module.
        for index in 1 .. 8
        loop

          Memory_request(
              request_type_formal      => receive_datum_octet,
              chunk_of_address_formal  => dont_care_X_datum,
              octet_formal             => octet_register);

          case index is
            when 1 => inm_max_packet.lo            := octet_register;
            when 2 => inm_max_packet.hi            := octet_register;
            when 3 => inm_address_length           := octet_register;
            when 4 => inm_time_out.lo              := octet_register;
            when 5 => inm_time_out.hi              := octet_register;
            when 6 => ack_type                     := octet_register;
            when 7 => local_net_type_of_service_table_row_size
                                                   := octet_register;
            when 8 => number_of_local_net_types_of_service
                                                   := octet_register;
          end case;
        end loop;

        -- Read in type of service translation table.

        declare
          row_number: integer range
                          0 .. number_of_local_net_types_of service;
          col_number: integer range
                          0 .. local_net_type_of_service_row_size;

          index:      integer range
                          0 .. number_of_local_net_types_of service
                             * local_net_type_of_service_row_size
                             := 0;
        begin
          row_number := 0;
          loop                      -- Outer loop reads all rows of TOS table.
            col_number := 0;
            loop                    -- Inner loop reads in a row of TOS table.
              Memory_request(
                  request_type_formal      => receive_datum_octet,
                  chunk_of_address_formal  => dont_care_X_datum,
                  octet_formal             => tos_table(index));

              col_number := col_number + 1;
              exit when col_number = local_net_type_of_service_row_size;

              index := index + 1;
              if  index > max_tos_table_size  then
                response := bad_srv_command;
                return;             -- Exit the current accept statement.
              end if;
            end loop;               -- End inner loop.

            row_number := row_number + 1;
            exit when row_number = number_of_types_of_service;
          end loop;                 -- End outer loop.
        end;                        -- End declare block.

      end Go;                                 -- End of init processing.

    end loop;                                 -- End of outer-most (infinite)
                                              -- loop.
end Read_Init_Parameters;
```

References

1. T. M. Carter, "ASSASSIN: An Assembly, Specification and Analysis System for Speed-Independent Control-Unit Design in Integrated Circuits Using PPL," Master's thesis, Department of Computer Science, University of Utah, June 1982.

2. A. L. Davis and P. J. Drongowski, "Dataflow Computers: A Tutorial and Survey," Computer Science Department Technical Report UUCS-80-109, University of Utah, Jul. 1980.

3. S. Hiyamizu et al., "Extremely High Mobility of Two-Dimensional Electron Gas in Selectively Doped GaAs/N-AlGaAs Heterojunction Structures Grown by MBE," *Japanese Journal of Applied Physics*, Vol. 20, No. 4, Apr. 1981, pp. L245-L248.

4. L. A. Hollaar, "Direct Implementation of Asynchronous Control Units", Submitted to IEEE Transactions on Computers, to be published December 1982

5. Organick, E. I., and Lindstrom, G., "Mapping high-order language units into VLSI structures," *Proc. COMPCON 82*, IEEE, Feb. 1982, pp. 15-18.

6. Postel, Jon: editor, "Internet Protocol: DARPA Internet Program, Protocol Specification," Tech. report RFC 791, Information Sciences Institute, USC, Sept. 1981.

7. C. L. Seitz, *System Timing*, Addison-Wesley, Reading MA, 1980, pp. 218-262, .

8. J. H. Shelly, "Design of Speed-Independent Circuits," File 226, UIDCL, July 1957.

9. H. E. Shrobe, "The Data Path Generator," *Digest of Papers: CompCon Spring 82*, IEEE Computer Society, 1982, pp. 340-344.

10. K. F. Smith; T. M. Carter; and C. E. Hunt, "Structured Logic Design of Integrated Circuits Using the Stored Logic Array," *IEEE Transactions on Electron Devices*, Vol. ED-29, No. 4, April 1982, pp. 765-776.

11. P. A. Subrahmanyam, "Automated Design of VLSI Architectures: Some Preliminary Explorations", Draft Version

Automated Design of VLSI Architectures: Some Preliminary Explorations

P.A.Subrahmanyam and S.Rajopadhye
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

Abstract

We discuss the design of a program transformation system that is geared to aid in the automated design of special purpose architectures (circuits), given a high level specification of a problem. The synthesis of systolic implementations is outlined, and examples of syntactic forms that aid in the description of such architectures (and algorithms tailored to them) are given. Finally, we summarize the results of applying the methodology in synthesizing several classes of systolic designs (proceeding from abstract, axiomatic specifications), and in the VLSI implementation of an Ada program fragment describing a part of the DoD Internet Protocol.

Research sponsored by Defense Advanced Research Projects Agency, US Department of Defense, Contract No. MDA903-81-C-0414

Table of Contents

1. Introduction

The need for design methodologies for special purpose VLSI circuits that help combat the spiralling complexity and cost of current day integrated circuit designs is by now well established [15]. We believe that it is also important that such methodologies enable a smooth embedding of the resulting circuits into larger systems that consist of both software and specialized hardware components e.g., on board control systems. In this context, we have been exploring the use of high level languages as a medium for specifying the desired behavior of special purpose systems, as well as paradigms for mapping such specifications into VLSI architectures [26, 25, 11]. We are currently developing a set of automated tools for transforming axiomatic and/or high level language program specifications in Ada into integrated software-hardware systems [12, 13]. In this paper we describe some of the details of the design of our transformation system, and in particular the manner in which the language constructs influence the architecture of the final machine. We then indicate some ways in which parallelism may be exploited, and how systolic designs may be synthesized. Syntactic constructs suitable for describing the behavior of special purpose architectures are also discussed. Finally, some preliminary results in applying the methodology to non-toy examples are outlined: these include various classes of systolic designs and a hardware implementation of an Ada program fragment that describes a part of the Department of Defense Internet protocol.

1.1. Overall Approach

We first summarize briefly our overall approach to the design of integrated software-hardware systems.

The initial specifications are annotated Ada programs. The "annotations" [9, 8, 22] allow for a statement of

1. Abstract axiomatic specifications of the behavior of a system, including statements of temporal characteristics.
2. Performance requirements to be met by an acceptable implementation along various dimensions of interest e.g., area, time, response time, throughput, reliability etc.
3. Relevant characteristics of the external environment a system is designed to operate in e.g., external timing constraints, relative function application frequencies, etc.

Given either abstract specifications, or an Ada program, or a combination, the following transformations may now be attempted:

- If the initial specifications are axiomatic, then these may be directly translated into an implementation suitable for being cast into silicon [25].
- Alternatively, the abstract specifications may be transformed into an implementation using primitives available in typical high level languages e.g., Ada [23].
- The high level language programs may be transformed into hardware implementations [12].

In essence, the annotated Ada specifications may be transformed into any desired mixture of software programs and special purpose hardware. The transformation into hardware is attempted in two phases: the output of the first phase is a symbolic description of the hardware implementation, which is then transformed into a set of masks suitable for actually fabricating the circuit. The latter translation uses a program that automatically generates layouts for asynchronous control units, given their symbolic description [3]; the layout of the data paths is currently done interactively using existing relatively low level design aids (e.g., a ComputerVision system).

The symbolic description of the hardware implementation is couched in an extended Ada syntax, by using "macros" for describing specialized hardware structures and algorithms tailored to them. Two major reasons for the use of such syntactic extensions are that (1) we have found it clumsy to describe certain kinds of concurrency (both at a high and low level) if we are constrained to use existing Ada program structures; (2) specialized primitives are very often more appropriate for succinctly describing algorithms that are tailored to special architectures.

We have found that the problem decomposition strategy and the configuration of target structures chosen is very often critically influenced by the desired performance requirements and the complexity measures associated with the target primitives available. This "strategy guidance" may be done either by using automated complexity computation aids [17, 24] or interactively.

A typical transformation scenario can roughly be divided into two "phases": (1) an *analysis* phase, wherein some global information relating to the program/specification is gathered; and (2) a *synthesis* phase wherein the implementation is built up. The analysis phase typically requires an examination of the *entire* program; this is usually done by traversing the parse tree. The synthesis phase is typically incremental in nature, and involves the use of the information gathered in the analysis phase and (optionally) further information of a more specific nature (i.e., not computed in the analysis phase) which may involve non-local analysis.

In essence, therefore, there is a common set of global properties needed for guiding the transformations which is profitably gathered in what we henceforth refer to as the (global) "analysis" phase, and a set of more specific properties that are better computed if and when needed. This separation into two phases, albeit somewhat nebulous, allows for

- —Conceptual clarity
- —Improved efficiency (because global traversals tend to be comparatively expensive)
- —Added flexibility in "global" decision making, since one is not forced to make an implementation decision too prematurely.

The remainder of this paper is organized as follows. In the next section we discuss the transformation of specific classes of syntactic constructs in Ada into hardware structures. In section 3, we focus on a few of the strategies useful that enable us to exploit parallelism, and then delineate the development of systolic designs (proceeding from either abstract specifications or from Ada programs). We describe some examples of syntactic constructs that aid in the succinct symbolic description of systolic designs, and in the transformation process. In appendix 1, we summarize the results of applying the methodology in the transformation of a fragment of an Ada program specifying the Department of Defense Internet Protocol [16] into a hardware implementation.

2. Transformation Strategies

We outline here a set of transformation strategies that we have developed for some of the commonly used syntactic constructs in Ada [1]. These can be broadly classified into either a "direct" (*in situ*) transformation of the language construct, or an "indirect" one, involving some optimization and flow analysis. The latter can be thought of as a set of source-to-source i.e., Ada-to-Ada transformations that account for the desired optimizations, followed by "direct" transformation. For the examples discussed in this paper, the target hardware model assumed is an asynchronous one [2] wherein state transitions controlled by request-acknowledge protocols that are implicitly embedded in the underlying model.

To facilitate exposition, we consider the Ada constructs in order of increasing complexity so that we can use the examples for, say, an assignment statement, in an if statement. We split the basic constructs into two classes. The declarative constructs serve to determine the collection of registers, the storage elements and the data paths between them. We refer to this as the "environment" part of the chip. The statements in the body of the program determine the (ensemble of) state machine(s) that constitute the "control" part of the chip. It is to be noted that this distinction is not very rigid, since, in general, the environment part of the circuit is affected by the statements and other constructs present in the procedural part of the program, and vice-versa.

The statement part of a program may in turn be viewed as contributing to either intertask communication or intratask computation. We envision an Ada task as a "standalone" circuit which is capable of communicating with other (co-)tasks. Since the Ada language specification does not detail the manner of this intertask communication, except for asserting that the underlying machinery ensures the existence of an asynchronous protocol (where the selection of ready tasks may at times be non-deterministic), we in fact implement an explicit interfacing machine which handles communication with other tasks. Its purpose includes maintaining information about the availability of the task machine for calls from the "outside", its allocation to different callers depending on any priority mechanism that might be desired, and maintaining queues to allow for conflicts. A detailed discussion of various intertask communication strategies and the trade-offs involved is contained in a companion report (see also [12]).

2.1. Declarative Constructs

2.1.1. Object Declarations

There are two kinds of object declarations in the Ada language — those which declare identifiers to be of a predeclared subtype, and those that declare them to be arrays. Of the predeclared subtypes, the most basic are the language-defined primitive subtypes, integer, real and boolean. A declaration of an identifier (or an identifier list) to be of any of these types results in its implementation being selected from a library of available primitives. For integers this presently consists of registers and RAM's. The registers used for integer implementation are in turn made up of flip-flops varying in complexity from simple flip-flops to two-phase, read-write flip-flops. The choice among these alternatives depends on the results of global data-flow analysis. Reals are implemented as special floating point registers, along with an encoding scheme and special arithmetic functions. Some booleans, depending on the results of global analysis may be found to be redundant in the circuit. These may result in their being implemented as combinational circuitry that computes their value at all instants. The booleans that cannot be eliminated in this manner are implemented as single flip-flops.

If the object declaration is an array declaration, this is usually implemented as RAM's of the appropriate primitive type. The range of values that the variable can assume is used to compute a default maximum size for the RAM which is further narrowed down, if possible, by using global analysis.

For object declarations that declare identifiers to be of some non-primitive type, the transformation system implements them as specified in the implementation of the type declaration for the particular type.

2.1.2. Type Declarations

An Ada type declaration defines a new class of objects. This can either be a simple range restriction on the predefined Ada types viz. integers and reals, an enumeration type, an array type definition, an access type definition, a derived type definition or a private type definition. For every type definition the transformation system maintains information about a default implementation in a predetermined template. When transforming object declarations of this type, this information is used to guide the particular implementation strategy adopted. The stored information is incrementally refined when global analysis is performed on identifiers declared to be of the particular type. Currently this is specified interactively by the user.

If the type declaration is a restricted range on a predefined Ada type, the limits of the range are either constant or variable identifiers. The first case implies a direct upper limit on the size of all identifiers that are declared to be of the type, and this information is added to the template implementation. If the limits of the range are identifiers, the results of global data-flow analysis for the identifier are used to establish an upper bound on the range, and this information is stored in the template.

Alternatively an Ada type declaration may define arrays, enumerations, records and access types. Currently the default array implementation consists of either RAM's or ROM's. The ranges of the indexing variables determine the size of the RAM, and the range of the type of individual objects in the declared array govern the word-size of the RAM. Since determination of minimum storage at compile time is, in general, a computationally impossible task, we have a default maximum on the size. The transformation system finalizes this decision after interacting with the user. Sometimes the user is able to specify the sizes more restrictively than the system ever could because of a more thorough understanding of the program. This aspect of the transformation system is more critical than in conventional compilers, because of our special target medium.

For enumeration types, the transformation system determines a minimal binary encoding for the set of objects declared in a reasonably straightforward manner. All future references to these enumerated constants are translated to a reference to one or more of these encodings. The other type definitions are somewhat more complicated, but the underlying theme of determining a default implementation for them is carried over, and a template is maintained to hold this information.

2.1.3. Renaming, Use and With Declarations

Renaming (and equivalent use/with) declarations are used in Ada to provide new names for identifiers, particularly if the identifier is declared in a different program unit. They do not imply that a separate copy is maintained, but are simply a notational convenience. As far as the chip architecture is concerned, they indicate the necessity of running a bus between two modules to make the variable available to both. (It is also possible to have duplicate copies, and ensure that consistency is maintained, but this approach is not currently used by the transformation system.) In cases where the whole circuit occupies more than one chip, or when the modules are physically placed far apart, renaming declarations enable some flexibility in exactly which module contains the actual instance of the object declared. (We currently prefer to rely more on use/with declarations, since too heavy a use of renaming declarations leads to more human errors that are not so easy or impossible for a compiler to detect.)

2.1.4. Subprogram, Package and Task Declarations

These kinds of declarations have been grouped together because, in general, they are all program units. Thus they indicate the presence of different computational modules. The scoping rules of Ada determine how these modules access variables present in other modules, and govern the generation of additional communication circuitry if necessary. If a subprogram module has more than one potential calling module it becomes necessary to provide some arbitration between possible conflicts. This is currently done interactively, where the user either specifies the arbitration circuitry or (usually) tells the system to assume that no conflict will occur.

2.2. Imperative Constructs

2.2.1. Assignment Statements (involving simple variables)

The general form of an assignment statement is

```
<Identifier> := <Expression>
```

The "code" for the target machine is generated by a top-down traversal of the parse tree. The transitions in the asynchronous target machine coincide with the order of node-visits in the top-down traversal of the parse tree. We illustrate the method with the familiar example as shown below. Consider the simple assignment statement

```
a := b*2-4*a*c;
```

with the abstract parse tree as shown below (Figure 2-1).

The root of this tree is mapped into a state, DoAssignment, which sends requests to subordinate states which perform the computations required. When it receives acknowledge signals from all such secondary states it causes the result to be "load"ed into the LHS of the statement. Here the code for "computing" the LHS is trivial since the LHS of the statement is a

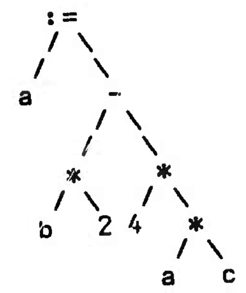

Figure 2–1: Abstract parse tree

simple variable. The only other subordinate state consists of ComputeRightArgument which computes the right–hand–side of the statement. Thus when the assignment node is visited, the system, recognizing that this is an assignment, issues a call to generate "code" for the "ComputeLeftArgument" and the "ComputeRightArgument" states. The code generated as a result (in the DoAssignment state) is to perform a request/acknowledge signal protocol with these two states, followed by code to achieve the actual load. For the "ComputeRightArgument" state, however, the code generated would await a request from "DoAssignment", issue calls to states that compute its arguments, await acknowledgements from them, and then (since the system knows that it is at a subtract node) perform the subtraction. This is independent of whether, say, the subtraction circuitry has been optimized to a series of decrements, or implemented by some general subtraction hardware. The state then returns an acknowledge to the "DoAssignment" state. This results in the following "code" for the "DoAssignment" state.

```
task body StateDoAssignment is

    begin
      accept ForkToComponents do
        fork (on (ReqToAssignment), to(StateComputeLeftArg,
                        StateComputeRightArg));
      end ForkToComponents;
    end StateDoAssignment;
```

The code for "join"ing the results of the two "fork"s and actually loading the results is as follows.

```
task body StateFinishAssignment is

    begin
      accept Dojoins() do
        join(on(Acknowledge), LastOfLeftArg,
                        LastOfRightArg);
      end Dojoins;
      hold(RegA.load);
    end StateFinishAssignment;
```

Continuing with the above example (for asynchronous implementation), we get the naive implementation shown in Figure 2–2. However, the transformation system is aware that parallelism can be successfully exploited to compute both operands of the subtraction simultaneously. Two of the states in the two separate "fork"s can be combined into one state, as can the 'shift-a' and 'shift-b' operations can be performed in just one state. The optimized state-machine is also shown in Figure 2-2. We also know that a can be left-shifted "in-place" because it is going to be modified later on in the computation. However, b cannot, and it must be "right-shifted" back to its original value. This is done in the last stage.

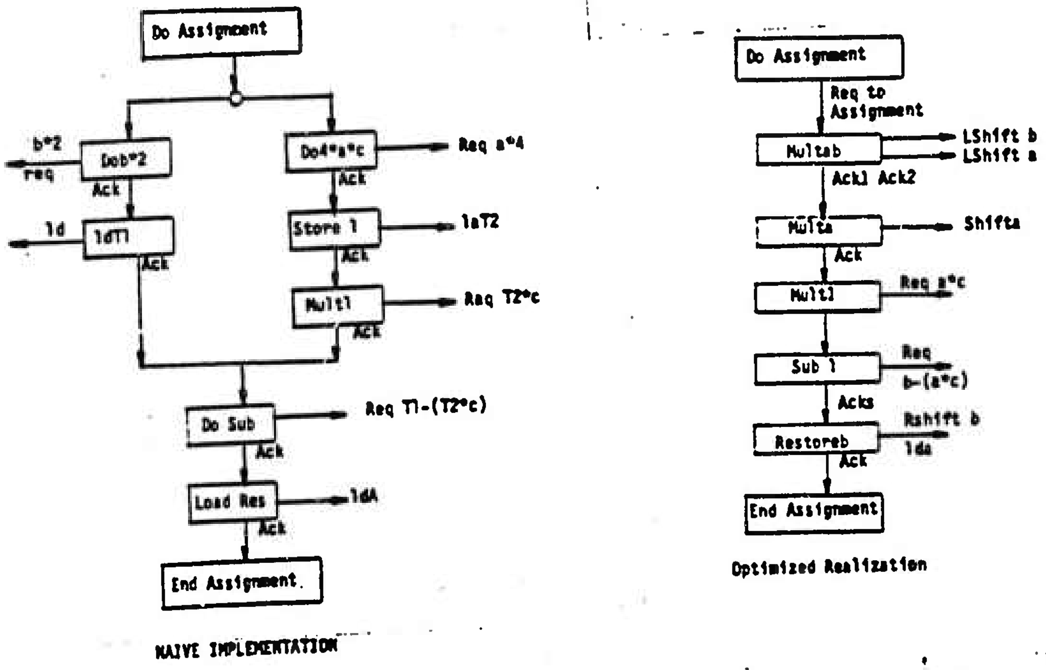

Figure 2-2: Two implementations of a:= b*2-4*a*c;

The graphical notation used here is a very useful aid to visualization and/or communication for human interaction, but is unnecessary as far as the transformation system is concerned.

Optimization as in the above example is made possible by means of data-flow analysis, as indicated earlier. We give some examples of the classes into which such information falls into.

1. Arithmetic Operations: Addition/Subtraction by 1 and sometimes 2 can be usually implemented as increment/decrements rather than using special arithmetic hardware. Multiplication/division by powers of two are cheaper when implemented as shifts. Boolean operations can be easily implemented directly as combinational circuits. Exponentiation, however, poses a problem because it invariably indicates the need for special hardware. An alternative strategy for implementing arithmetic operations is to have a special module that is used by more than one state-machine. Decisions such as these typically involve consideration of trade-offs (viz space vs speed) and a study of frequency and regularity of usage is required in this context.
2. Common-Subexpression Identification: This is done using standard data-flow analysis techniques. Elimination of such portions of the program involves either source-to-source transformation, or some kind of "indirect" transformation of the source to incorporate the results of the data flow analysis.
3. Temporary Storage Determination: This entails an analysis of the requirements for storing intermediate results. Data flow analysis and work concerning optimization of temporary allocation in conventional compilers is useful in indicating if registers (or other storage) used in earlier parts of the machine can be reused. Here we have an added advantage in that there is no a priori upper bound on the number of such storage units, and they are not restricted to hold only certain specific data types.

2.2.2. Conditional Statements

The general form of conditional statements is as follows.

```
<ConditionalStatement> := 'if' <Condition> 'then' <Statement>
                          ['else' <Statement>] .
```

The machine that performs an if statement's function consists of a state (or a set of states) to evaluate the <Condition> part. This state returns a boolean value, depending on which the machine makes a transition to one of two states that are the start states for the two <Statement>'s. If there is no else clause, one of the branches makes a transition directly to the statement following the if statement. This is shown in the figure below (Figure 2-3).

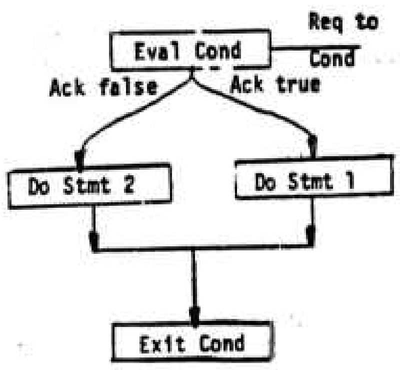

Figure 2-3: Skeletal state machine for an if statement

In addition to guiding the transformation of assignment statements, inferences from global analysis as are used to determine the presence of redundant boolean variables in the source program. Such variables are then replaced by just the output line from some combinational circuitry.

2.2.3. Loop Statements

Ada provides for both simple, unconditional loops as well as while, for, and until loops. A construct of the form

```
'loop' <SequenceOfStatements> ;
```

is implemented as the set of states that execute the <SequenceOfStatements>, followed by a direct transition to the first state in the <SequenceOfStatements>. Any "exit" statements inside the <SequenceOfStatements> translate to transitions to the state immediately following the loop.

We indicate in the next section how such constructs may be used for the synthesis of systolic chips.

For while-loops of the form

```
'while' <Condition> 'loop' <SequenceOfStatements> ;
```

the transformation is similar, with the exception that the states for <SequenceOfStatements> are preceded by states similar to those for a conditional statement (without an else clause), and the last state in <SequenceOfStatements> is followed by an unconditional transition back to the states for evaluating the condition.

For constructs wherein the loop consists only of a select statement, (many task bodies fall into this category,) the loop can be replaced by a single state where the machine waits until it receives a signal from any of the modules that call the corresponding accept statements. It then makes a transition to the appropriate set of states, performs the required computation, and returns to the "wait" state.

2.2.4. Procedure Calls

Procedure calls are directly implemented using Request/Acknowledge communication between the caller and the state machine that implements the procedure. The state machine first loads the parameters of the procedure on the bus/lines to the called machine, and then issues a request to it. Alternatively, a "lazy evaluation" kind of scheme may be used, where the parameters are evaluated by the caller only when needed by the called module (in response to a demand from it.) After the caller receives the acknowledge from the function module (which implies that the output data line(s) from it are valid) it makes a transition to the next state.

Global analysis is used to obtain information such as the following:

1. Whether it is useful to implement the function "in line". This saves some communication overhead at the expense of increased silicon area. In effect such an arrangement provides a private copy of the procedure to every caller. In VLSI we have the added advantage of not being restricted to a universal scheme. Some procedures can be implemented in-line while others may be centrally shared modules. An even more general solution provides some callers (depending on estimated/measured frequency of use) with private copies of the function, while others share a common unit.
2. Identification of globals accessed in the procedure body. This involves deciding on appropriate communication protocols and routing considerations.

2.3. Optimization

Optimizations of a design are possible at all of the levels in the design hierarchy:

—At the very lowest level, it is possible to increase system performance by redesigning individual transistor layouts (e.g. changing Width/Length ratios) to increase speed etc.

—At a somewhat higher level, performance improvements can be obtained by using specialized circuits to achieve certain functions instead of using a standard cell set.

—At the next level, symbolic version of layouts can be locally "manipulated" in order to improve efficiency e.g., this may involve swapping adjacent columns (or rows) of PPLs etc., while ensuring that logical function is not impaired.

—At the state machine level, performance improvement can affected by state minimization, improved parallelism, etc.

—Finally, the high level architecture of the implementation can be juggled in order to improve performance, while maintaining consistency with the the abstract, representation independent, specifications of the problem.

It is important to note that these levels have rough analogs in the realm of standard language translation/machine architecture: faster/more powerful instruction sets, peephole optimization, flow analysis on intermediate compiler code, and algorithm improvement. Further, the overall improvement is typically greater the closer the optimizations are to the initial stages of development of an implementation: it is therefore more advantageous to attempt to design an appropriate architecture (/algorithm), rather than spend time optimizing channel layouts.

3. Systolic Architectures

In this section we delineate a few transformations that enable the synthesis of some classes of systolic designs. For the sake of brevity, we deal here only with a few classes of looping and recursion constructs. The methods are applicable to a wider class of starting points, and the theoretical basis for the mechanical synthesis of such designs (among others) is elaborated upon in [26]. As a consequence, we have here chosen to emphasize examples of syntactic constructs that are suitable for describing such algorithms and architectures, rather than the details of the synthesis strategy itself.

The primary decompositions possible are one or more of ***sequential composition, parallel decomposition***, and ***pipelining***. Which decomposition scheme is adopted typically depends upon the performance criteria desired, a detailed analysis of which we omit here. For example, pipelining improves throughput, while parallel processing improves both throughput and response time over sequential solutions. Of course, the response time is very much dependent upon the algorithm used (i.e., upon what the specific decomposition is, what the subcomputations involved are, and how the partial results are combined), and to a lesser extent upon the lower level circuit implementation strategies. In particular, we recall that as a consequence of wire delays being the dominant factor in single chip implementations, asynchronous implementation strategies are preferable in order not to slow down the whole system and to minimize skewing effects.

We now discuss examples of syntactic macros that aid the representation of such decompositions.

3.1. Iteration

Consider the loop structure

```
for i in 1 .. N loop
        x(i) := F(x(i))
end loop;
```

A possible sequential implementation of this loop structure is shown in Figure 3-1. This implementation consists of a processing element (or cell) that computes the function F. When the stream of values $x_1, ..., x_n$ is input to the F-cell, the output is the stream $F(x_1), ... F(x_n)$.

A parallel implementation is possible if the computation of F does not have any side effects on the subsequent computations in the loop. Such an implementation can use N instances of the same F-cell, input the vector of values $<x_1,...,x_n>$ in parallel, and output the vector of results $<F(x_1),...,F(x_n)>$ in parallel. The i-th instance of the F-cell thus inputs x_i and outputs $F(x_i)$. This is illustrated in Figure 3-2.

When each computation through the loop results in the computation of a partial result that is "assembled together" in the subsequent iterations, a pipelined implementation can be generated.

Thus, if we consider

```
for i in 1 .. N loop
        x := F(x)
end loop;
```

then a pipelined implementation using N instances of F-cells is shown in Figure 3-3.

A combination of one or more of these techniques can obviously be employed whenever needed.

3.2. Recursion

Some classes of recursive functions (procedures) can also be mapped into systolic implementations. It is of course possible to first apply standard recursion to iteration transformations and then apply the techniques discussed here. It is however also possible to avoid this intermediate step in several cases. As an example, the form shown below can be directly transformed into either of the implementations shown in figure 3-4.

```
function Match(s, p: string) return boolean is
begin
        if s=null
```

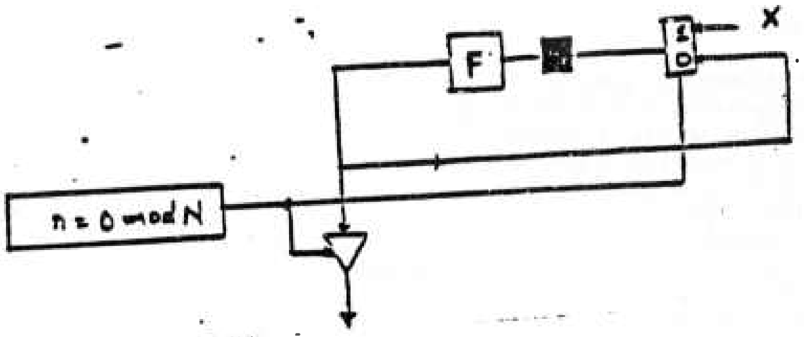

Figure 3–1: Sequential Implementation

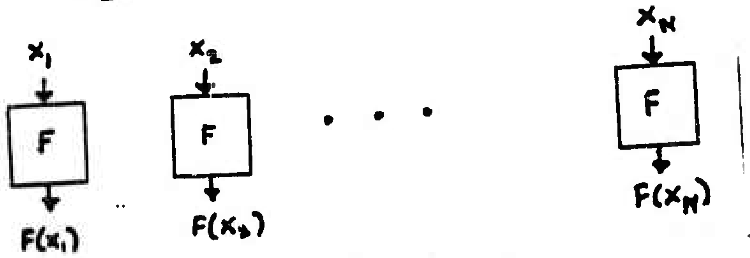

Figure 3–2: Parallel Implementation

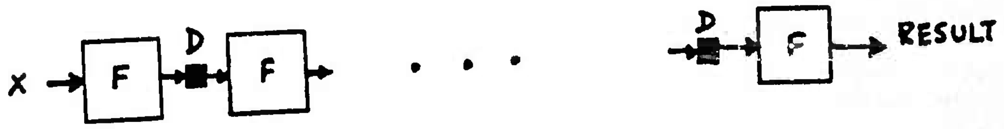

Figure 3–3: Pipelined Implementation

```
then if p=null
        then return(true)
        else return(false)
else if Last(s)=Last(p)
        then Match(All_But_Last(s), All_But_Last(p));
end Match;
```

3.3. Syntactic Constructs for Expressing Systolic Designs

We now outline examples of syntactic forms for expressing some of the commonly occurring features of systolic algorithms e.g., regular interconnection patterns, modes of progress of data streams, and the *loci of computations* (a mathematical generalization of the notion of "wave fronts" of a computation introduced in [26]). The details of the syntactic forms given here are by no means unique or final, and will doubtless undergo change as our experiments progress. Our objective is mainly to illustrate how specialized "derived" or "macro" forms may be developed to suit specific interconnection topologies at hand. The presence of such macros some important

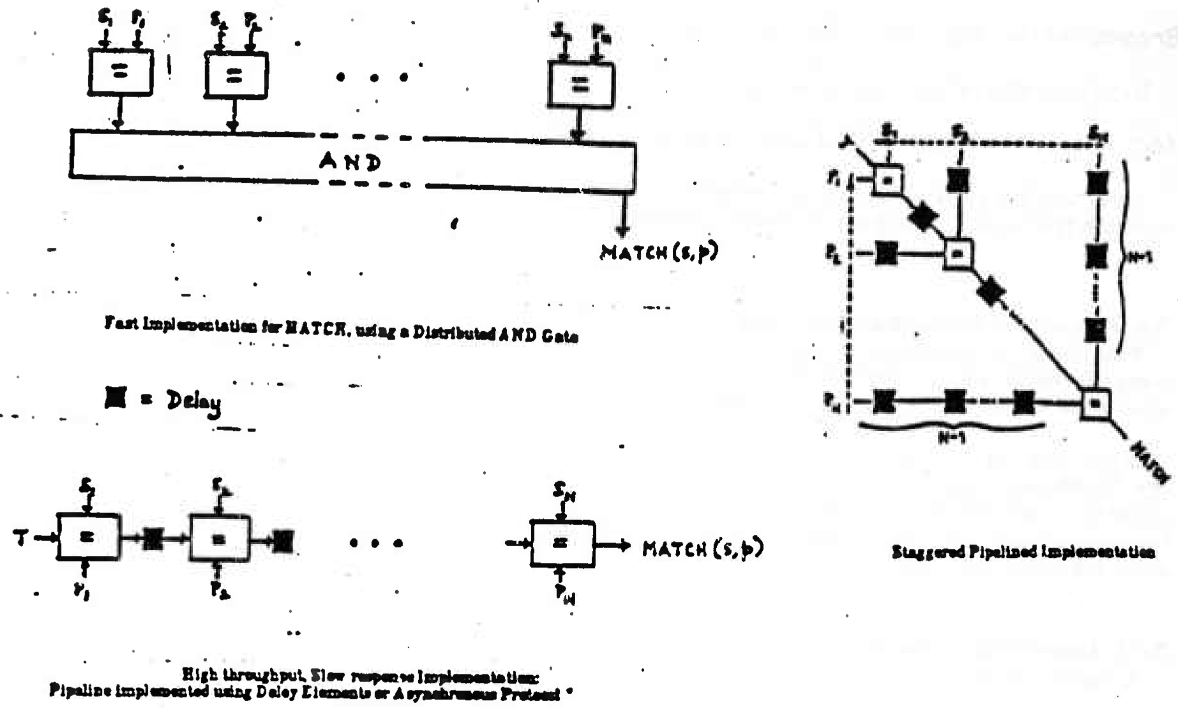
AND
MATCH(s,p)
Fast Implementation for MATCH, using a Distributed AND Gate
= Delay
T
MATCH (s,p)
High throughput, Slow response Implementation:
Pipeline implemented using Delay Elements or Asynchronous Protocol
N-1
N-1
MATCH
Staggered Pipelined Implementation

Figure 3-4: Implementations generated for Match

advantages over the expanded/graphical forms in pattern matching and automated transformation in that (1) there is a significant decrease in complexity in doing textual pattern matching over doing graphical pattern matching (sublinear vs. quadratic or more); (2) the absence of global inter-dependency of subcomputations in the iteration body is explicit and does not have to be *inferred* by global data flow analysis; (3) performance metrics can be easily defined over such succinct representations: this facilitates automated complexity computation, although a graphical representation (which is isomorphic) typically facilitates human computation/comprehension.

3.3.1. Broadcasting

Broadcasting a signal to a set of ports associated with some collection of processing elements is stated as

```
Broadcast(signal, Set_of_Ports)
```

For example, the Set_ of_ Ports may be a collection of named ports of an array of similar processing elements.

Roughly speaking, port names of cells may be viewed as entries of tasks associated with them. Thus, consider a MULTIPLY_ ADD_ CELL that accepts has 3 inputs (ports) a, b, and c, and outputs a single value a*b+c. We can describe a linear array of MULTIPLY_ ADD_ CELL's which is useful in several systolic algorithms for matrix computations, as

```
MULTIPLY_ADD_CELLS: array(1..N) of MULTIPLY_ADD_CELL;
```

If we then want to state that x is broadcast to the N input ports named "a" of the array of processing elements MULTIPLY_ ADD_ CELLS, we can express this as

```
Broadcast(x, MULTIPLY_ADD_CELLS.a)
```

Note that this is identical to saying

```
for i in 1..N loop CONNECT(x,MULTIPLY_ADD_CELLS(i).a) end loop;
```

This can be generalized in an obvious manner to more complicated cases, including one wherein the set of ports is computed dynamically.

3.4. Regular Interconnection Structures and Related Operations

When a set of processing elements have regular interconnections with their neighbors, it aids comprehension and pattern matching if the "local" and "global" parts of the interconnections are stated succinctly (as opposed to specifying the detailed interconnections).

While the components of an architecture is described by the set of interconnections between the hardware modules it consists of, its functioning, or the computational details of an algorithm tailored to it involves stating how input data streams move through the system, get operated upon, and ultimately emerge as output streams. We now give examples of these in some standard settings.

3.4.1. Linear Interconnections

A pipelined computation in linear interconnection of a set of cells can be expressed as

```
Pipeline(Array, Direction, BoundaryConditions,
                  Set_of_Output_Ports, Set_of_Input_Ports_of_Adjacent_Cell)
```

where Direction is either left–to–right or right–to–left, the BoundaryConditions state what is input at the left or right extreme port and what is to be done at the corresponding output, and the *pair* of sets Set_ of_ Output_ Ports and set–of–input–ports specify the set of complementary port names that detail which ports of adjacent cells are interconnected.

As a specific example, we have

```
Pipeline(MULTIPLY_ADD_CELLS, LeftToRight, 0,
                    MULTIPLY_ADD_CELL(i).c, MULTIPLY_ADD_CELL(i+1).a)
```

or

```
Pipeline(MULTIPLY_ADD_CELLS, LeftToRight, 0,
                    MULTIPLY_ADD_CELL.c, Right(MULTIPLY_ADD_CELL).a'
```

where Right(MULTIPLY_ ADD_ CELL) indicates the cell to the right of the current cell in the linear array.

Such constructs can be generalized. As an example, we next consider tree interconnections.

3.4.2. Tree Interconnections

As an example, we give the skeletal specification of the operations and workings of a "Dictionary machine" that has the main computation performed by its leaf processors. Note that the broadcasting process may itself be defined in terms of a task (in Ada).

```
task Dictionary

        entry INSERT(k: in KEY; r: in RECORD);
        entry DELETE(k: in KEY);
```

```
          entry SEARCH(k: in KEY; r: out RECORD);
          entry UPDATE(k: in KEY; r: in RECORD);
          entry MIN_RECORD(r: out RECORD);
          ...
   end;

   task body Dictionary is

          TREE: BinaryTree(DictionarySize, LeafProcessor,
                                InternalNodeProcessor);
                 -- This processor tree implements the Dictionary
                 -- LeafProcessor and InternalProcessor are 2
                 -- types of processors ("task types") that are
                 -- used in instantiating the tree.

          FunctionPort, KeyPort, RecordPort: Port;
                 -- FunctionPort represents the physical lines
                 -- that activate the function invoked and the
                 -- lines needed for the request/acknowledge protocol.
                 -- KeyPort and RecordPort represent the physical
                 -- lines associated with k and r.

          -- The association between the logical ports and physical ports
          -- is detailed below. The general form of this construct is
          -- REPRESENT(physical-port-name, function-name, parameter-name)
          -- whi   states that the "physical-port-name" represents
          -- the "parameter-name" associated with "function-name".
          -- This enables statement of time multiplexing of the lines.

          REPRESENT(KeyPort, INSERT, k);

          ...
          REPRESENT(KeyPort, MIN_RECORD, k);

          REPRESENT(RecordPort, INSERT, r);
          REPRESENT(RecordPort, SEARCH, r);
          REPRESENT(RecordPort, UPDATE, r);
          REPRESENT(RecordPort, MIN_RECORD, r);

          ....

          -- The interconnections to the global ports are described below

          CONNECT(Root(TREE).ANSWER, RecordPort);
          CONNECT(Root(TREE).KeyPort, KeyPort);
          CONNECT(Root(TREE).FunctionPort, FunctionPort);
          ...

   begin
      loop
          select
                 accept SEARCH(k: in KEY; r: out RECORD) do
                        Broadcast(k, r, Leafs(TREE).SEARCH);
                               -- delay O(log(DictionarySize))
                               -- this is done by making use
                               -- of the internal node processors.
                               -- The "Answer" from the root is
                               -- connected to the global port
                               -- corresponding to r.
                 end SEARCH;

                 ....

          end select;
      end loop;
   end Dictionary;
```

```
   task type LeafProcessor is
```

```
        entry INSERT(k: in KEY; r: in RECORD);
        entry SEARCH(k: in KEY);

        ...

end LeafProcessor;
task body LeafProcessor is

        LeafKey: KEY;            -- This is the local key.
        LeafRecord: RECORD;      -- This is the record in the
                                 -- leaf processor, or a code
                                 -- indicating that there is no
                                 -- record at this leaf.


begin
   loop
        select
               accept SEARCH(k: in KEY) do
                      if LocalKey = k and DEFINED(LocalRecord)
                      then Father.ANSWER(LocalRecord);
                      ...
               end SEARCH

               ....

        end select;
     end loop;
 end LeafProcessor;
--------------------------------------------------------------------------

task type InternalNodeProcessor is
         entry INSERT(k: in KEY; r: in RECORD);
         entry DELETE(k: in KEY);

         ...
         entry ANSWER(AnswerFromLeftSon, AnswerFromRightSon: in RECORD;
                      CombinedAnswer: out RECORD);
end  InternalNodeProcessor;

task body InternalNodeProcessor is
begin
         loop
                 select
                          accept SEARCH(k: in KEY; r:out RECORD) do
                                 Broadcast(k, Sons.SEARCH);
                                 delay 1;
                                 ....

                          end SEARCH

                          ....

                 end select;
         end loop;
 end Dictionary;
```

3.5. Input and Output of Data Streams

The manner in which the data needed for a computation is input and output is of great importance in designing highly concurrent algorithms. Succincts descriptions of such data streams is therefore important, and also serves a dual purpose in aiding simulations much the same way as do I/O drivers in lower level circuit simulations.

```
A: array(1..N) of BITS;
```

```
INPUT(A);                -- Represents an array of bits input in parallel
INPUT(SKEW(A,1));        -- Represents a rotated wavefront of bits
                         -- where A(1) is input at time 1,
                         --       A(2) is input at time 2, ...
                         -- and so on.
```

Additional timing statements may similarly be incorporated. These forms can be expanded in a natural manner to help describe the operation of algorithms tailored to specific architectures e.g. [10].

3.6. Distribution of Data. A Systolic Stack Implementation

Given an abstract specification, one of the important decisions in developing highly concurrent designs relates to how the "data" components can be/are implemented in a distributed fashion. In [26], we have developed a general technique for aiding such decisions, which has been applied to the development of hardware implementations for matrix manipulations, stacks, prinority queues, adders, symbol tables and a few graphics related algorithms. Here, we illustrate how the result of the synthesis for a Stack may be couched syntactically using the forms we have outlined above.

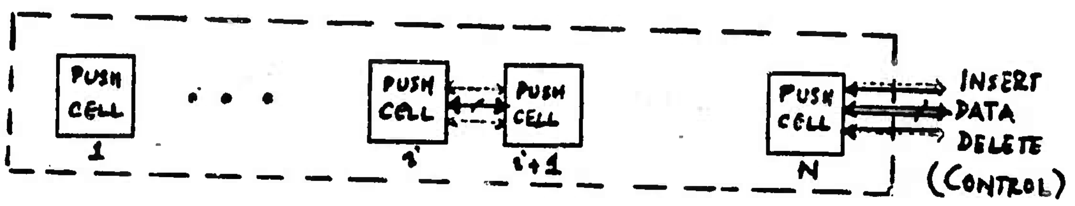

Figure 3-5: A Systolic Stack Implementation

```
task SystolicStack is
        entry INSERT(x: in Element);
        entry DELETE(x: out Element);

        -- Additional "semantic annotations" specify the behavior
        -- of these to be that associated with the abstract data
        -- type "Stack". We omit these here for brevity.

end SystolicStack;

task body SystolicStack is

   record
        CellArray: array(1..N) of PushCell
        ...
   end;

-- Local Interconnections

        Connect(PushCell.SendLeft, Left(PushCell).INSERT);
                                   -- this is a Right To
                                   -- Left data transfer direction

        Connect(PushCell.GetFromLeft, Left(PushCell).DELETE);
                                   -- this is a Left to Right
-- Boundary Conditions

        Connect(INSERT, CellArray(N).INSERT);
        Connect(DELETE, CellArray(N).DELETE);

        Connect(CellArray(1).GetFromLeft, UNDEFINED);
        Connect(CellArray(1).GetFromLeft, 0);

begin
   loop
     select
        accept INSERT(x: Element) do
                CellArray(N).INSERT(x);
        end INSERT;
        accept DELETE(x: out) do
                CellArray(N).DELETE(x);
        end DELETE;
     end select;
   end loop;

end SystolicStack;
--------------------------------------------------------------------------------

task type PushCell is
        entry INSERT(x: in Element);
        entry DELETE(x: out Element);
        entry SendLeft(x: out Element);
        entry GetLeft(x: in Element);
end;

task body PushCell is

        CurrentElement: Element;

begin
        accept INSERT(x: Element) do
                OutputToLeft(SendLeft(CurrentElement));
                                        -- transmitting current contents
                                        -- to left neighbor
                                        -- Both these operations can be
                                        -- done in 1 cycle using a
                                        -- 2-phase clocked flip-flop
                CurrentElement:=x;

        end INSERT;

        ...
        -- We omit implementations for the other ports.

end PushCell;
```

3.7. Some Remarks

We have outlined in this section how some classes of iterative and recursive constructs may be transformed to yield systolic designs. In addition, we have given examples of syntactic constructs that can be used to succinctly describe regular interconnection patterns, data streams, and (the progress of) loci of computations. The exact syntactic forms described are merely examples of "macro forms" that can be defined in terms of other extant constructs; they are neither unique nor final, and will doubtless undergo change as our experiments progress. The presence of such forms facilitates comprehension, reduces the complexity of pattern matching needed to perform automatic transformations, and also reduces greatly the amount of data flow analysis that needs to be performed in order to achieve the same results.

The details of the transformation strategy for a more general class of specifications, and a mathematical basis for supporting such automated synthesis may be found in [26]. The discussion there also elaborates on how the performance criteria, cost metrics and technology constraints affect the synthesis strategies.

4. Conclusions

We have detailed in the preceding sections the structure of an automated transformation system geared to aid in designing systems that consist of a mixture of software components and special purpose (VLSI) hardware components. In particular, we have indicated the mapping of various syntactic constructs in Ada into hardware structures, and some other high level constructs into systolic implementations. It is intended that these transformation tools be based on the theoretical framework developed in [26], and therefore produce designs that are formally verifiable.

An additional contribution has been to delineate syntactic forms that aid succinct descriptions of special purpose architectures and algorithms tailored to them. The design of such constructs has been done to aid direct mapping into circuit layouts, and to reduce the complexity of pattern matching involved in the transformation process. Such forms may be in fact be viewed as "macros", since they may be elaborated using the existing set of Ada primitives. Unfortunately, however, the resulting expansions are sometimes quite clumsy and obfuscating; on the other hand, a potential use of these expansions is in simulation of the resulting hardware using commercially available compilers for Ada.

Finally, we have summarized some of the results of our preliminary empirical explorations in using the transformation/synthesis methodology. The examples considered included various classes of systolic algorithms and the hardware implementation of an Ada program fragment using "path programmable logic" [20, 14]. Our preliminary results have been quite encouraging, and have served to emphasize the importance of performance characteristics in determining the global synthesis strategy. It has been estimated that the trade-off in using the latter methodology for low level VLSI design results in about 10-20% increase in chip area required (when compared with custom layouts), but results in a drastic reduction in the design time (from a few months to a few days) [20].

Acknowledgements. We gratefully acknowledge the feedback received on various aspects of this work from our colleagues in the "Ada-to-Silicon project", particularly Elliott Organick, Tony Carter, Al Davis, Alan Hayes and Gary Lindstrom. Special thanks go to S.Purushothaman for porting the transformation system to run on the Vax.

Appendix

1. Hardware Implementation of a part of the Internet Protocol: A Case Study

In this appendix, we summarize the results of applying the methodology detailed above in the transformation of a fragment of an Ada program specifying the Department of Defense Internet Protocol [16] into a hardware implementation. The Internet Protocol (henceforth referred to as IP) is a communication protocol designed to enable packets to be transferred *between* networks. The function of the particular module that we consider here (called Read_ Init_ Parameters) is to read in the initialization parameters from the Memory Unit, and to send an acknowledgement to the caller when it is done. The procedure shown below (Figure 4-1) is a general procedure that achieves this while admitting a great deal of flexibility in the sizes of various parameters.

Generation of the Circuit for Read_ Init_ Parameters

The Ada program shown above is transformed using the methodology outlined earlier. For the most part, it corresponds to a direct application of the strategies outlined in 2. Some of the salient features resulting from the optimizations are described below.

The case statement, which constitutes the major portion of the <SequenceOfStatements> part of the first loop is very highly specialized in that it simply checks the index variable of the loop and, depending on its value, chooses a variable that is loaded from a specific variable (octet_ register). As a result this is implemented by using a multiplexor which is controlled by the loop variable.

Since the variable "number_ of_ tos_ table_ octets" is the product of two variables "local_ net_ type_ of_ service_ row_ size" and "number_ of_ local_ net_ types_ of_ service", and is never used except in a final escape clause in the second loop, we use two nested loops and do away with the multiplication altogether.

The final target code is shown below. A symbolic description of the circuit obtained from this by using the Assassin program [3] and laying out the data paths is also shown here. This form of the circuit can be directly transformed into a set of masks for fabrication, an instance of which is also shown.

```
separate (Inm_Out_Module.Inm_Out)

procedure Read_init_parameters(response: out out_response) is

  procedure Memory_request(
      chunk_of_address_formal:      chunk_of_address_type;
      do_write_formal:              boolean;
      octet_formal:              out octet_type)

          renames Memory.Request;

  octet_register:                octet;

begin

  -- Download the 6 individual initialization parameters.

  for index in 1 .. 8
  loop
    Memory_request(
       request_type_formal      => receive_datum_octet,
       chunk_of_address_formal  => don't_care_X_datum,
       octet_formal             => octet_register);
    case index is
      when 1 => inm_max_packet(8)             := octet_register;
      when 2 => inm_max_packet(1)             := octet_register;
      when 3 => inm_address_length            := octet_register;
      when 4 => inm_time_out(8)               := octet_register;
      when 5 => inm_time_out(1)               := octet_register;
      when 6 => ack_type                      := octet_register;
      when 7 => local_net_type_of_service_table_row_size
                                                 := octet_register;
      when 8 => number_of_local_net_types_of_service
                                                 := octet_register;
    endcase;
  end loop;


  number_of_tos_table_octets    :=
               local_net_type_of_service_table_row_size    *
                        number_of_local_net_types_of_service;

  for index in 1 .. number_of_tos_table_octets
  loop
    Memory_request(
       request_type_formal      => receive_datum_octet,
       chunk_of_address_formal  => don't_care_X_datum,
       octet_formal             => tos_table(index));
  end loop;

end Read_init_parameters;
```

Figure 4-1: Source Program for Read_ Init_ Parameters

```
with TransformationGenerics, NewBoolean;

procedure RIPTarget is

task RIPStart is
     entry ReqRIP;
end RIPStart;

task body RIPStart is

package InmMaxPacketLow           is new Register(size => 8);
package InmMaxPacketHigh          is new Register(size => 8);
package InmAddressLength          is new Register(size => 8);
package InmTimeOutLow             is new Register(size => 8);
package InmTimeOutHigh            is new Register(size => 8);
package InmAckType                is new Register(size => 8);
package InmTOSTableRowSize        is new Register(size => 8);
package NoOfLocNatTOS             is new Register(size => 8);
package TOSSizeCounterPrelimReg   is new Register(size => 8);
package TOSEntryCounter           is new Register(size => 8);
package EntryDone                 is new EqComparator(size => 8);
package TOSDone                   is new EqComparator(size => 8);
package Loop1Decoder              is new EnDecoder(InputSize => 3);
package TypeOfServiceTable  is new RAM(AddressSize => 8,
                                       WordSize    => 4);
package TOSAddressRegister  is new CirIncRegister(size => 8);

package ControlUnit is

 task RIPState1 is
        entry Move2;
 end RIPState1;

 task RIPState2 is
        entry Move3;
 end RIPState2;

 task RIPState3 is
        entry Move1;
        entry Move4;
 end RIPState3;

 task RIPState4 is
        entry Move5;
 end RIPState4;

 task RIPState5 is
        entry Move6;
 end RIPState5;

 task RIPState6 is
        entry Move7;
 end RIPState6;

 task RIPState7 is
        entry Move5;
        entry Move8;
 end RIPState7;

 task RIPState8 is
        entry Move5;
        entry MoveSTRT;
 end RIPState8;


 task body RIPState1 is
        begin
          accept Move2() do
            move(on(MemoryRequest.Ack), to(RIPStateS2));
          end Move2;
          hold(MemReq);
        end RIPState1;

task body RIPState2 is
        begin
          accept Move3() do
            move(on(NIL), to(RIPStete3));
          end Move3;
          reset(MemReq);
          hold(Loop1Decoder.Enable)
        end RIPState2;

task body RIPState3 is
        begin
         select
          accept Move1() do
            move(on( NOT(MemoryRequest.Ack)) AND
                     NOT(DecoderCounter.Carry))),
                                      to(RIPState1));
          end Move1;

          accept Move4() do
            move(on( NOT(MemoryRequest.Ack) AND DecoderCounter.Carry),
                                      to(RIPState4));
          end Move4;
          hold(TOSSizeCounterPrelimReg.Inc);
        end RIPState3;

task body RIPState4 is
        begin
          accept Move5() do
            move(on(NIL), to(RIPState5));
          end Move5;
          hold(TOSSizeCounterPrelimReg.Clr);
          hold(TOSEntryCouvnter.Clr);
          hold(TOSAddreeeRegieter.Clr);
        end RIPState4;

task body RIPState5 is
        begin
          accept Move6() do
            move(on(MemoryRequnet.Ack), to(RIPState6));
          end Move6;
          hold(MemReq);
        end RIPState5;

task body RIPState6 is
        begin
          accept Move7() do
            move(on(NIL), to(RIPState7));
          end Move7;
          reset(MemReq);
          hold(TypeOfServiceTable.Write)
        end RIPStata6;

task body RIPState7 is
        begin
          accept Move8() do
            move(on(NIL), to(RIPState8));
          end Move8;
          hold(TOSSizeCounterPrelimReg.Inc);
          hold(TOSEntryCounter.Inc);
          hold(TOSAddreeeRegister.Inc);
        end RIPState7;

task body RIPState8 is
        begin
         select
          accept Move5() do
            move(on(NOT(TOSDone)), to(RIPState5));
          end Move5;
          accept MoveSTRT() do
            move(on(TOSOone), to(RIPStart));
          end MoveSTRT;

          hold(TOSSizeCounterPrelimReg.Clr);
          hold(TOSEntryCounter.Inc);
        end RIPState8;

end ControlUnit;

               begin -- body of task RIPStart

                     accept ReqRIP do
                         move(on (InmServer.Request), to (RIPStateS1));
                     end ReqRIP;
                     hold(TOSSizeCounterPrelimReg.Clr);

               end RIPStart;


begin -- Body of procedure RIPTarget, specification of
      --                 interconnections

 CONNECT (MemoryRequest.Output(0..7), TypeOfServiceTable.Input(0..7));
 CONNECT (MemoryRequest.Output(0..7), InmMaxPacketLow.Data(0..7));
 CONNECT (MemoryRequest.Output(0..7), InmMaxPacketHigh.Data(0..7));
 CONNECT (MemoryRequest.Output(0..7), InmAddressLength.Data(0..7));
 CONNECT (MemoryRequest.Output(0..7), InmTimeOutLow.Data(0..7));
 CONNECT (MemoryRequest.Output(0..7), InmTimeOutHigh.Data(0..7));
 CONNECT (MemoryRequest.Output(0..7), InmAckType.Data(0..7));
 CONNECT (MemoryRequest.Output(0..7), InmTOSTableRowSize.Data(0..7));
 CONNECT (MemoryRequest.Output(0..7), NoOfLocNetTOS.Data(0..7));

 CONNECT (EntryDone.Input1(0..7), InmTosTableRowSize.Data(0..7));
 CONNECT (EntryDone.Input2(0..7), TOSSizeCounterPrelimReg.Data(0..7));

 CONNECT (TOSDone.Input1(0..7), NoOfLocNetTOS.Data(0..7));
 CONNECT (TOSOone.Input1(0..7), TOSEntryCounter.Data(0..7));

 CONNECT (TypeOfServiceTable.Address(0..7), TOSAddressRegister);

 CONNECT (Loop1Decoder.Input(0..2),
              TOSSizeCounterPrelimReg.Data(0..2));
 CONNECT (Loop1Decoder.Output(0), InmMaxPacketLow.Load);
 CONNECT (Loop1Decoder.Output(1), InmMaxPacketHigh.Load);
 CONNECT (Loop1Decoder.Output(2), InmAddressLength.Load);
 CONNECT (Loop1Decoder.Output(3), InmTimeOutLow.Load);
 CONNECT (Loop1Decoder.Output(4), InmTimeOutHigh.Load);
 CONNECT (Loop1Decoder.Output(5), InmAckType.Load);
 CONNECT (Loop1Decoder.Output(6), InmTOSTableRowSize.Load);
 CONNECT (Loop1Decoder.Output(7), NoOfLocNetTOS.Load);

end RIPTarget;
```

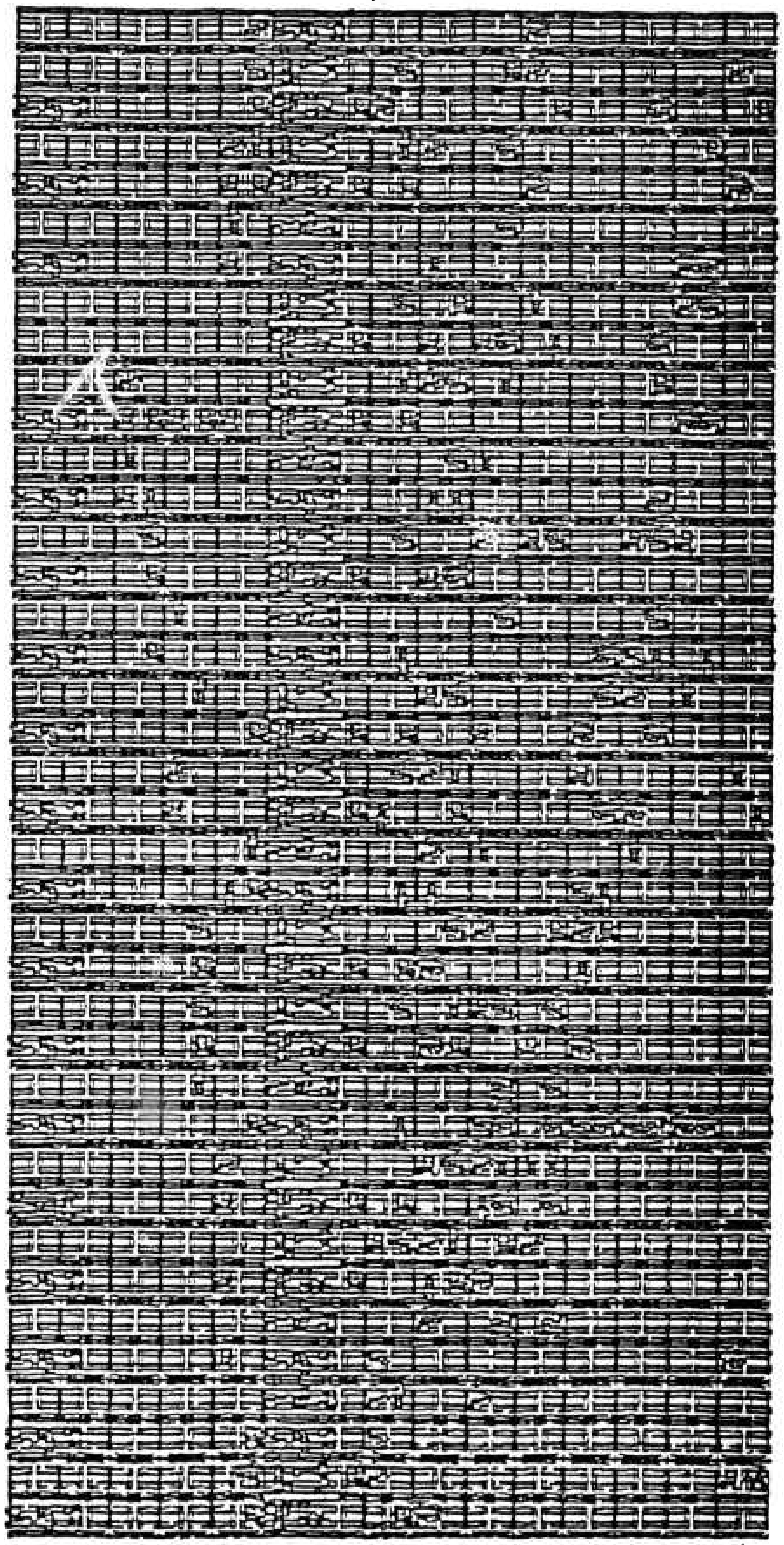

Figure Composite Layout (NMOS) for Read_ Init_ Parameters Control

References

1. *Reference Manual for the Ada Programming Language, Proposed Standard Document.* July, 1980 edition edition, United States Department of Defense, 1980. For sale at U.S. Government Printing Office, Order No. L008-000-00354-8.

2. T. M. Carter and L. A. Hollaar. Implementation of Asynchronous Control Unit State Machines in Integrated Circuits Using the Storage/Logic Array (SLA). Available From the VLSI Research Group at the University of Utah

3. T. M. Carter. ASSASSIN: An Assembly, Specification and Analysis System for Speed-Independent Control-Unit Design in Integrated Circuits Using PPL. Master Th., Department of Computer Science, University of Utah, June 1982.

4. L.J.Guibas and F.Liang. Systolic Stacks, Queues and Counters. Proc. 1982 Conf. on Advanced Research in VLSI, M.I.T., 1982, pp. 155-165.

5. L.J.Guibas, H.T.Kung, and C.D.Thompson. Direct VLSI Implementation of Combinatorial Algorithms. Proc. of the Caltech Conference on VLSI, January, 1979.

6. Heller, D.E. and I.C.F. Ipsen. Systolic Networks for Orthogonal Equivalence Transformations and Their Applications. Proceedings, Conference on Advanced Research In VLSI, MIT, The Pennsylvania State University, University Park, PA, January, 1982, pp. 113-122.

7. Johnsson, L. A Computational Array for the QR-method. Proceedings, Conference on Advanced Research in VLSI, MIT, California Institute of Technology, Pasadena, CA, January, 1982, pp. 123-129.

8. B.Krieg-Bruckner and D.Luckham. ANNA: Towards a Language for Annotating Ada Programs. Proc. of the ACM-SIGPLAN Symposium on the Ada Programming Language, Boston, Mass., SIGPLAN, December, 1980, pp. 128-138.

9. Krieg-Bruckner, B., Luckham, D.C., von Henke, F.W., Owe, O. (Draft) Reference Manual for Anna, A language for Annotating Ada Programs. Unpublished, Reviewer's Copy, October 1982.

10. Kung, S.Y., Gal-Ezer, R.J., Arun, K.S. Wavefront Array Processor: Architecture, Language, and Applications. Proceedings, Conference on Advanced Research in VLSI, MIT, USC, January, 1982, pp. 4-20.

11. Organick, E. I., and Lindstrom, G. Mapping high-order language units into VLSI structures. Proc. COMPCON 82, IEEE, Feb., 1982, pp. 15-18.

12. Organick, E.I., Carter, T.M., Lindstrom, G., Smith, K.F., Subrahmanyam, P.A. Transformation of Ada Programs into Silicon. SemiAnnual Technical Report. Tech. Rept. UTEC-82-020, University of Utah, March, 1982.

13. Organick, E.I., Carter, T., Hayes, A.B., Lindstrom, G., Nelson, B.E., Smith, K.F., Subrahmanyam, P.A. Transformation of Ada Programs into Silicon. Scond SemiAnnual Technical Report. Tech. Rept. UTEC-82-103, University of Utah, November, 1982.

14. Patil, S. S. and Welch, T. "A Programmable Logic Approach for VLSI." *IEEE Trans. on Computers C-28* (Sept 1979), 594-601.

15. J. G. Peterson. Keys to Successful VLSI System Design. Proceedings of the 1981 CMU Conference on VLSI Systems and Computations, Computer Science Department, Carnegie-Mellon University, October, 1981, pp. 21-28.

16. Postel, Jon: editor. Internet Protocol: DARPA Internet Program, Protocol Specification. Tech. Rept. RFC 791, Information Sciences Institute, USC, Sept., 1981.

17. Ramachandran, R. A Complexity Computation Package for Data Type Implementations. Master Th., University of Utah, Department of Computer Science, June 1982.

18. Seitz, C.L. Ensemble Architectures for VLSI — A Survey and Taxonomy. Proceedings, Conference on Advanced Research in VLSI, MIT, California Institute of Technolog, Pasadena, CA, January, 1982, pp. 130–135.

19. Shrobe, H.E. The Data Path Generator. Proceedings, Conference on Advanced Research in VLSI, MIT, Cambridge, Mass, January, 1982, pp. 175–181.

20. K. F. Smith; T. M. Carter; and C. E. Hunt. "Structured Logic Design of Integrated Circuits Using the Stored Logic Array." *IEEE Transactions on Electron Devices ED–29*, 4 (April 1982), 765–776.

21. Edward A. Snow III. *Automation of module set independent register transfer level design.* Ph.D. Th., Carnegie–Mellon University, April 1978.

22. Subrahmanyam, P.A. From Anna+ to Ada: Automating the Synthesis of Ada Package and Task Bodies. Tech. Rept. Internal Report, University of Utah, March, 1982.

23. P.A. Subrahmanyam. An Automatic/Interactive Software Development System: Formal Basis and Design. In H.J–Schneider and A.I.Wasserman, Ed., *Automated Tools for Informations System Design and Development*, North–Holland, Amsterdam, 1982.

24. Subrahmanyam, P.A. On Automating the Computation of Approximate, Concrete, and Asymptotic Complexity Measures of VLSI Designs (to appear). Tech. Rept. UTEC–82–095, Dept. of Computer Science, University of Utah, October, 1982.

25. Subrahmanyam, P.A. Abstractions to Silicon: A New Design Paradigm for Special Purpose VLSI Systems. Tech. Rept. UTEC #82–065, University of Utah, January, 1981 (Revised May 1982). Submitted for Publication to TOCS

26. Subrahmanyam, P.A. An Algebraic Basis for VLSI Design. Draft of a Research Monograph, April 1982. Available from the Department of Computer Science, University of Utah.

27. Wile, Dave. POPART: A Producer of Parsers and Related Tools, System Builder's Manual. Unpublished, USC/ISI

UTEC-82-016

ADA TO SILICON TRANSFORMATIONS: THE OUTLINE OF A METHOD

by

Lawrence A. Drenan[1] and Elliott I. Organick

Dept. of Computer Science
University of Utah
Salt Lake City, Utah 84112

This research was sponsored in part by the Defense Advanced Research Projects agency, DARPA contract No. MDA903-81-C-0414.

September 1982

[1]Presently employed by Western Digital Corp, 2445 McCabe Way, Irvine, CA 92715

ABSTRACT

This report explores the contention that a high-order language specification of a machine (such as an Ada program) can be methodically transformed into a hardware representation of that machine. One series of well-defined steps through which such transformations can take place is presented in this initial study.

The general method consists of a two-fold strategy:

1. Transform the high-level specification into a network of inter-communicating "state machine/data path pairs".

2. Through a catalogue method, map each state machine / data path pair into a circuit realization.

Four representational levels are utilized in the transformation process. Each inter-level transformation is discussed. The four levels are:

1. Ada specification of the algorithm.

2. Machine-description specification of the algorithm, consisting of a control part and a data part. This version is expressed in a stylized dialect of Ada developed for this study.

3. Protocol-definition specification of the algorithm, obtained by inserting constructs that define inter-program unit communication.

4. Storage/Logic Array (SLA) specification of the algorithm, which can be mapped directly to, and are regarded as equivalent to, circuit representations.

The transformation strategy relies upon exploiting a one-to-one correspondence between Ada instantiations of generic packages introduced in the level 2 representation and SLA "modules", which are composed of primitive SLA cells introduced at level 4.

The transformation methodology described in the paper has been demonstrated for a non-trivial Ada program example.

1. Introduction

This report reviews elementary principles applicable for methodically transforming a high-order language specification of a machine, such as an Ada program, into a hardware representation of that machine. In this initial study, we discuss one series of well-defined steps through which such transformations can take place.

Research on automating Ada-to-Silicon transformations is currently underway at the University of Utah [9]. In this report, which does not attempt to document the specifics of the mainstream of that research, we outline a series of mappings for transforming individual Ada program units to equivalent integrated circuits. Our emphasis is on the feasibility of these transformations and is not concerned with finding a series of <u>optimal</u> transformation steps. Our purpose is to:

1. Demonstrate one (relatively straightforward) approach by which an Ada program can be mapped into a specification of an integrated circuit (IC) through adherence to rule-based techniques.

2. Examine the pros and cons inherent in the most straightforward, unoptimized approach.

The method presented follows the general transformation strategy suggested earlier [8]. The essence of this strategy is to represent each Ada program unit as a synchronous stored state machine part and a data path part. Circuits derived by following this approach have the general form pictured in Figure 1-1. The pairing of a state machine and a data path (i.e., an environment) is referred to as an "engine". The hardware realization of an entire Ada program, or of any subset of program units of that program, is actually a network of asynchronously intercommunicating engines, each having the form outlined in Figure 1-1. For the convenience of this report, individual Ada tasks are considered to be program units.

A transformation methodology is just beginning to be explored [11]. There is need to develop a well-defined set of rules through which such transformations can eventually become a mechanical process. Some guidelines that distinguish a set of rules as having the potential for eventual automation have been suggested [10].

```
                                             Input
                                               |
                                               v
                                        ***************
*****************        control        *    Local    *
* State Machine *---------------------->* Environment *
*      Part     *                       *    Part     *
*****************                       ***************
       ^                                    |      |
       |              feedback              |      v
       --------------------------------------     Output
```

Figure 1-1: An Engine and Its Two Principal Components

The transformations presented here are considered to be extensions of those originally outlined in the following sense:

1. Not only is the high-level specification of a program unit expressed in Ada; intermediate levels of representation are also expressed in Ada. "Machine-description" and "Protocol-definition" styles of Ada programming are proposed to express intermediate transformation steps, permitting the algorithmic behavior to be checked through Ada program execution at all intermediate levels as well as the top level.

2. NMOS Storage Logic Array (SLA) technology [15] [14] is chosen for the low-level realization of the machine. (More practical versions of SLAs, called <u>PPLs</u> have been developed to serve as a target for this transformation process [9].) SLA "modules" give us a set of building blocks that fit the specific needs of this method. Utilization of other semi-custom integrated circuit components offers an opportunity for enrichment of this methodology into the VLSI range.

A high-order language Ada program is transformed in three steps to reach the level of representation from which integrated circuits may be produced directly. In this report, the four levels, counting the starting level, are called "stages". These stages are:

1. High-level Ada program

2. Machine-description Ada program

3. Protocol-definition Ada program

4. NMOS SLA program or equivalent

Characteristics of these stages and rules that guide the transformations between them are presented in succeeding sections. A case study that was performed following this method on a non-trivial Ada program is presented elsewhere [6].

[We again stress that circuit optimization (space or speed) is not a goal addressed in this paper. Thus, in situations where performance or circuit area or both are critical, the approach presented is unlikely to yield circuits with characteristics that are competitive with those produced by more custom methods, especially for many important, but special algorithms, e.g., those that lead to compact systolic arrays.]

2. Stage 1: High-Level Ada Program

The machines specified and realized by our transformation process are viewed as ensembles of interacting state machine/environment pairs (engines). The programming language Ada is well-suited for specifying such pairs. Thus, a strong correlation exists between data abstractions in Ada and data abstractions in certain views of integrated circuits; indeed we exploit this correlation.

An Ada program is composed of one or more program units [5] [2]. A program begins execution as a single thread of control in the main subprogram, but can initiate tasks, each of which has associated with it a separate thread of control. A program unit in this model is analogous to a machine that is initiated via a single "Go" button, but which is capable of delegating work among potentially concurrent sub-machines. In Ada, such sub-machines take the form of <u>tasks</u>. Ada also offers flexibility and control in specifying the communication between program units, i.e., in specifying the kind of interaction between units. Data abstractions represented as Ada <u>packages</u>, another form of program unit, are also transformable into individual engines whose operators either transform given instances of a data type or own and operate on individual instances. Shifting such an engine from idle to a particular active state corresponds, at a higher level of abstraction, to the activation of an Ada package operation.

Information needed to represent an engine can be extracted from an Ada program unit for use in representing the local environment (data path) and the state machine (controller). This information is drawn both from the specification part and from the body part of the program unit being mapped to the next stage.

Stage 2 representation elaborates intra-program unit constructs while Stage 3 elaborates inter-program unit communication constructs. The language for Stage 2 is a stylized but legal form of Ada.

3. Stage 2: Machine-description-level Ada program

3.1. The Role of Stage 2

A Stage 2 program achieves two objectives:

1. Infers a collection of needed hardware modules from the declaration part of the program unit and identifies the needed modules through instantiation of generic packages.

2. Transforms infix expressions represented in the Stage 1 form into prefix form.

The distinction between the control flow and data flow of a program is sharpened by the transformation from Stage 1 to Stage 2. Thus, in its Stage 2 form, the program takes the form of a state machine and the data path it controls. The declarative part of the Stage 2 form represents a collection of hardware modules (a "data path") inferred from the declarative part of the Stage 1 form. The body part of the Stage 2 form represents a state machine whose structure is inferred from both the declarative and body parts of the Stage 1 form. The Stage 2 language style has two distinguishing features:

- extensive use of generic building blocks

- use of the "engine extension" style of representing states and state transitions

The terms "building block" and "module" have specific meanings below. A "building block" refers to a generic package instance introduced in Stage 2 to model a particular component of the data path. A "module" refers to a collection of SLA cells from which the full circuit will be constructed. Every generic package instance identified in the Stage 2 representation maps to a corresponding Stage 4 SLA module.

3.2. Stage 2 Examples

Figure 3-1 is an example of a generic package declaration for a building block representing a counter. An instantiation of this package (e.g., "package C is new Counter") corresponds to the module's "black box" representation (see Figure 3-2). The SLA program that corresponds to Figure 3-2 is presented in Figure 3-3.

```
generic
  lo_value: integer;
  hi_value: integer;
             -- allows one to instantiate
             -- counters of various sizes
package Counter is
    -- Function:
    --   a counter with load, lookup,
    --   increment, and decrement operations
  procedure Load(
      load_value: in integer );
  procedure Increment;
             -- Increment by 1 is implied.
  procedure Decrement;
             -- Decrement by 1 is implied.
  function Lookup return integer;
             -- Returns the current value.
end Counter;
```

Figure 3-1: Counter Building Block Package Specification

With a few exceptions (to be discussed below) all variables and operators in the Stage 1 program unit are transformed into instantiations of generic

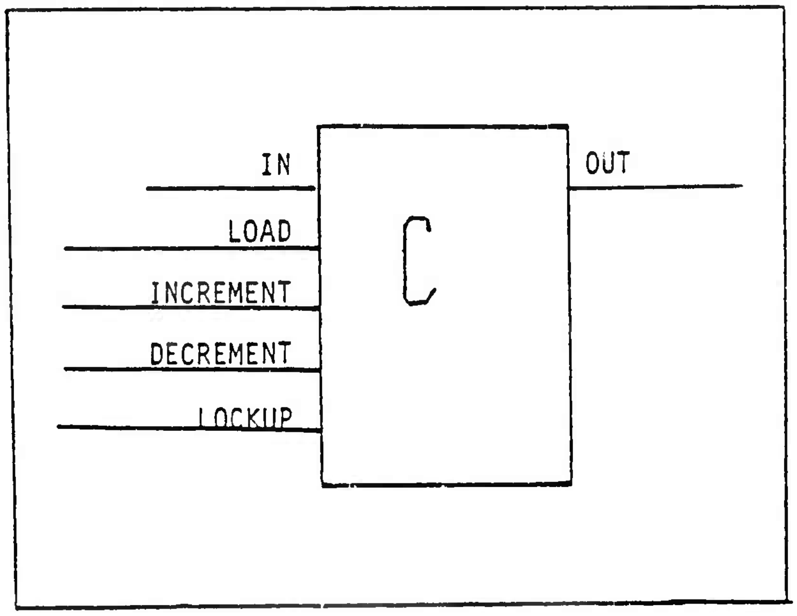

Figure 3-2: "Black Box" Representation of a Counter Module

packages. The Stage 2 code is then restricted to describing actions through the use of these instantiated packages. Stage 1 to Stage 2 transformations result in code that is composed primarily of function and procedure applications. For example, a line of code such as

```
A := B + C;
```

is transformed into

```
A.Write(Add.Go(B.Read, C.Read));
```

where A, B, C, and Add are previously instantiated packages. Thus, if the Stage 1 code includes the object declaration

```
A, B, C: integer;
```

the corresponding Stage 2 form would exhibit the instantiations

```
package A is new Register(word_length => integer);
package B is new Register(word_length => integer);
package C is new Register(word_length => integer);
```

```
    0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
    1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
 1:   _   _   _   _ 0 B 0 B 0 B 0 B 0 B ¦;
 2:   "   "   "   "                     ¦;
 3:   "   "   "   "                     ¦;
 4:   "   "   "   "                     ¦;
 5: F B F B F B F B         1$          ¦;
 6:                 + 1     +           ¦;
 7:                 +   1   +           ¦;
 8:                 1$     =+=1$        ¦;
 9:                =+     1$" "         ¦;
10:                 "       " "         ¦;
11:  $R " " " " " " " 0 0 1     0$      ¦;
12:  $S " " " " " " " 0 0 1     1$      ¦;
13:      $R " " " " " 0 0 1       0$    ¦;
14:      $S " " " " " 0 0 1       1$    ¦;
15:          $R " " " 0 0 1         0$  ¦;
16:          $S " " " 0 0 1         1$  ¦;
17:              $R " 0 0 1           0$¦;
18:              $S " 0 0 1 " "= = = =1$¦;
19:            $0 S " 1 0 0 " "$¦ " " " ";
20:        $0 S 1 R " 1 0 0 " "$¦ " " " ";
21:    $0 S 1 R 1 R " 1 0 0 " "$¦ " " " ";
22:$0 S 1 R 1 R 1 R " 1 0 0 " "$¦ " " " ";
23:$1 R 1 R 1 R 1 R " 1 0 0 " "$¦ " " " ";
24:            $1 R " 0 1 0 " "$¦ " " " ";
25:        $1 R 0 S " 0 1 0 " "$¦ " " " ";
26:     1 R 0 S 0 S " 0 1 0 " "$¦ " " " ";
27: 1 R 0 S 0 S 0 S " 0 1 0 " "$¦ " " " ";
28:=0=S=0=S=0=S=0=S "=0=1=0=" "$¦ " " " ";
29: _ _ _ _ _ _ _ _ _ _ _ _ _ _ _  " " " ";
```

Figure 3-3: SLA Program for Counter Module Using the SCLED Notation

Furthermore, encountering "+" while parsing Stage 1 code would lead to the inclusion of

```
    package Add is new Adder;
```

in the corresponding declarative part of the Stage 2 code. Hence, the code presented in this example would eventually map into a hardware structure abstractly presented in Figure 3-4.

The design of the building block set and the design of the SLA module set must be coordinated. As a possible means of enforcing the design discipline, a Stage 2 programmer is provided with one or more packages that specify the set of

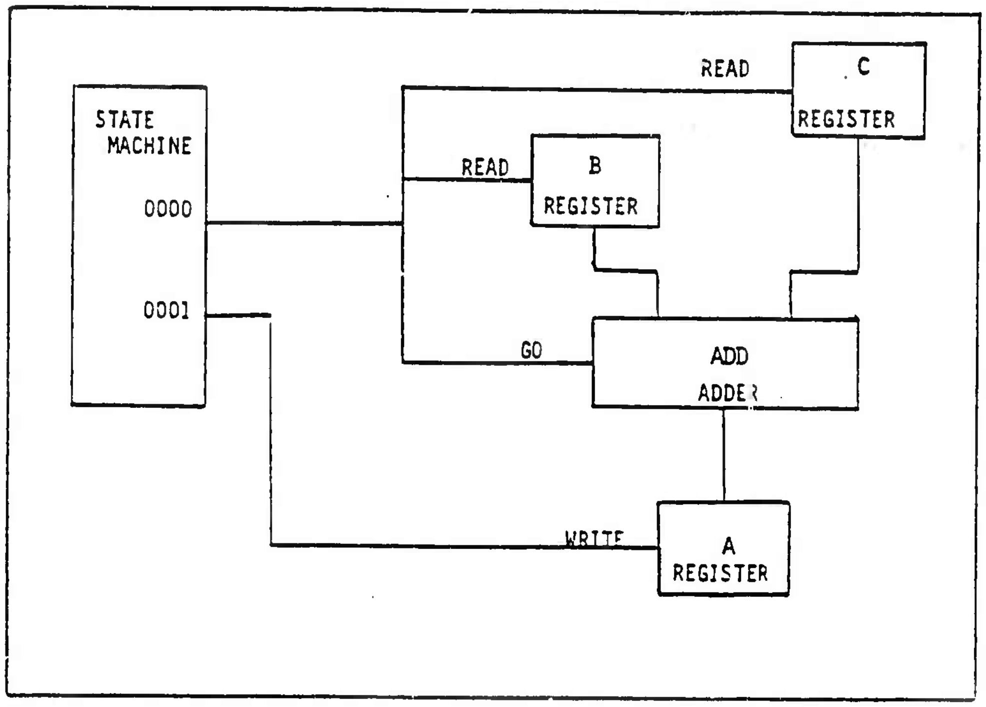

Figure 3-4: Hardware Realization of "A := B + C;"

generic packages available. The programmer can thereby be restricted to expressing algorithms with instantiations and use of the pre-defined generic packages.

3.3. The "Engine" Extension to Ada

The body part of a Stage 2 program is sub-divided into states denoted by labels. To represent the mutually independent actions that can occur in the same state of a state machine in standard Ada, one could use the "verbose form" that declares (and then initiates) a set of dynamically created tasks. A more succinct equivalent is possible if we were to include an "engine extension" for Ada to specify a similar objective. Used at Stage 2, the engine extension allows one to specify a sequence of Ada statements that can be translated into concurrent actions.

An engine clause has the structure illustrated in Figure 3-5. Within the scope of an engine clause, the sequence of statements bounded by two state

```
engine Example is
  begin
    <<State_Start>>   -- initial actions
                      -- executed in parallel


    <<State_1>>       -- actions to be
                      -- executed in parallel

    <<State_2>>       -- another set of actions which
                      -- can be executed in parallel

    <<State_stop>>    -- final state
                      null;
  end Example;
```

Figure 3-5: Structure of an Engine Clause for Representing "Transition Graph" of a State Machine

labels, e.g., <<State_1>> and <<State_2>> above, are actions that can occur in parallel. Execution of a "goto" statement within such a (labeled) sequence terminates the actions within that state (i.e., triggers a state transition). (To enhance readability, we follow the convention that the first node of every engine clause be labeled "State_Start" and the final node be labeled "State_Stop".)

Nesting of engines clauses follows Ada scoping rules. An engine may be declared local to another engine just as one procedure can be declared local to another procedure. Thus a local "sub-engine" may be called from its containing "main-engine". The effect of such a call is to transfer control to the label State_Start of the subengine at the time the subengine is called and to return control to the main engine when the subengine completes.

Note that this technique does not imply a relationship between state transitions and units of time. Although the particular SLA implementation chosen for Stage 4 in this work is synchronous, a syntax comparable to the engine extension has been be mapped to asynchronous implementations [4]. An algorithm used to determine the operations for which one can specify parallel execution, i.e., multiple actions within the same state, is presented in Section

5.

3.4. Building Blocks and Modules

For the purpose of this report, the following building blocks and modules have been designed [6]: Equals, Less_eq, Bool_eq, Counter, Loop_Counter, Register, Boolean_Register, Memory, and Two_D_Memory.

Building blocks and modules generally have parameters for specifying word lengths. Such specifications are provided by the Stage 2 programmer as part of an interactive design process. Thus, most generic package declarations contain the formal generic parameter

```
        type word_length is range <>;
```

3.5. Three Intra-program Unit Communications Protocols

Three different intra-program unit protocols are defined, corresponding to the "function", "procedure", and "procedurE" Stage 2 subprogram declarations. These Stage 2 declarations convey assumptions about the number of states required for an operation to "complete its job". Different protocols may be utilized for invoking various operations within an implemented package. The corresponding SLA implementation is invoked with whichever protocol is appropriate. Protocols for communication between circuits representing separate Ada program units are discussed in Section 6.)

Operations are divided into two classes: those that return a value (e.g., a Read operation) and those that do not (e.g., a Write operation). Hardware implementation of the former requires that the module includes storage elements to hold the value of the output parameter (or function result). The protocols presented below ensure that such storage elements are sampled only after the correct values are loaded. In operations that do not return a value, the protocols ensure that the module completes its job (for example, modification of a global value) before a potentially conflicting operation can be initiated.

The distinguishing characteristics of operations adhering to each of the three protocols are as follows:

- "Function" protocol: The operation completes in the same state in which a request for the operation reaches the containing module. Two cases are implementable:

  1. The function result is always available.

  2. The request is received in phase Phi-1 of a given clock cycle, and causes the result to be available in phase Phi-2 of the same clock cycle.

  A function operation (such as the Lookup operation on a Counter module) does not need to issue an acknowledge to its requestor that it has performed its duty, because it can be assumed that the correct result will be available in a known state.

- "Procedure" protocol: The operation completes in the state immediately following the one in which the request reaches the module. As in the function protocol, it is not necessary for the procedure operation (such as the Increment operation on a Counter module) to inform the requestor that the desired action has been performed.

- "ProcedurE" protocol: For this operation, it cannot be assumed that the job will be completed in the same state in which the request is received, or even in the next state. Unlike the two previous protocols, it is necessary for the containing module to inform the requestor when execution of the desired action has been completed. The scenario is as follows: a requestor initiates a procedurE operation by issuing a "Go" signal; the procedurE in turn signals its caller, upon successful completion, with an "I'm done" signal. We call this convention the "Go/I'm done" protocol. Its use allows the introduction of arbitrary delays in the state transitions for clocked schemes that exhibit a single thread of control. The protocol, which is enforced by construction, is implemented as follows:

  * The requesting engine R sends a "Go" signal that invokes the type procedurE operation P of a containing module M and then enters a state where R waits for M to send an "I'm done" signal.

  * The initial state of M is a wait state for a "Go" signal. A Go for P causes the states the operation P to commence (transition to P). After the operation P completes M emits an "I'm done" signal before returning to its initial state.

The protocol permits representation of a single thread of control that traverses from the requesting engine R to the host module M of the procedurE operation P and back again. The sequence of state transitions for every procedurE operation is local to one, and only one, engine. Hence, there is no possibility for contention. It is this fact that allows us to use the simple "Go/I'm Done" protocol (instead of a somewhat more complex Request/Acknowledge) for intra-engine communication. The Read and Write operations on the Memory module are examples of the procedurE protocol.

4. Stage 1 to Stage 2 Transformations

4.1. Transforming Simple Expressions

Simple expressions are transformed in a straightforward way. Registers replace variables, comparators replace relational operators, adders replace plus signs, etc. Such transformations are syntax driven.

This style of transformation leads to the allocation of possibly redundant modules. Clearly, circuits produced by this method tend to be wasteful of "real estate". However, timing and communications are simplified in activating individual modules, since each Stage 2 call on a subprogram operation of a generic instantiation then corresponds to a unique control line in the hardware level. Some simple optimizations are possible within this framework; for example, use of counters where adders are not needed, and use of shift logic, where suitable, for multiplication or division.

4.2. Transforming Control Statements

The interpretation of control statements (e.g., loop, case, if, subprogram calls and task entry calls) lead to control flow changes. We discuss the required transformations for such constructs in this subsection on a case by case basis. In general, these transformations mimic well-understood strategies used by compilers [1].

Procedures, functions, and tasks The initial action to be performed in the body parts of procedure, function, and task entries with in parameters is the loading of the actual parameter values into the Registers that implement the corresponding formal parameters. Statements directing such actions must be inserted into the Stage 2 program.

Out parameters also require instantiation of Register packages so their values can be loaded into these Registers as if they were local parameters and hence mimic the "copy-restore" parameter passing mechanism demanded (for the normal case) by Ada semantics. A similar treatment is required so that function values can be properly returned.

Building blocks that represent formal parameters of program units are derived in Stage 2. For example, if procedure P and function F are specified as:

```
procedure P(
    xx: integer;
    yy: integer);
function F(
    zz: integer)
   return real;
```

then four generic packages are instantiated:

```
package xx is new Register(word_length => in integer);

package yy is new Register(word_length => in integer);
                                                    -- For P.

package zz is new Register(word_length => in integer);

package f_result is new Register(word_length => real);
                                                    -- For F.
```

<u>IF-STATEMENTS</u> In the simplest case, if-statements are manifested in Stage 2 as structures of the form:

```
<<State_for_if>>  if  condition  then
                    goto State_X;
                  else
                    goto State_Y;
                  end if;
```

Missing but implicit else clauses are explicitly inserted. For example:

```
                  else
                    goto State_<the_state_where_the_2_branches_join>;
```

It is certainly possible, and in many cases advisable, to include actions in the branches before the goto statement, thereby reducing the total number of states specified in the machine description. For example,

```
if  mem_value = 0  then
  pointer := p_find;
  exit;
end if;
```

is transformed into

```
declare
  equals_result: boolean := false; -- Initialized to
  ...                               -- false.
begin
  ...
<<State_4>>  Equals.Test(
                 Mem_value.Lookup(), 0, equals_result);
             goto State_5;

<<State_5>>  if  equals_result  then
               Pointer.Write(P_find.Lookup());
               goto State_6;          -- Goes to exit.
             else                     -- Else is now explicit
               goto State_7;
             end if;
```

Notice the use of the boolean variable "equals_result" to represent the value of the condition. The rule followed is that the use of identifiers with "_result" as a suffix specifies Stage 4 routing to a storage element that is located within the module specified by the prefix (e.g., Equals). The storage element is loaded with the result of the operation. Every relational operator building block has such a "buddy" boolean variable. Out parameters in procedures and procedurEs, such as the value returned from a memory Read procedurE, are also treated this way.

<u>BLOCKS</u> A block is treated as a parameterless procedure.

<u>FOR-LOOPS</u> A generic Loop_Counter package that computes and holds the loop parameter value is instantiated for each Stage 1 for-loop. This package also

stores the value of the upper limit of the discrete range. In case the upper bound is a previously declared variable, e.g., Lim, a module that stores Lim's value already exists, so the extra storage element is redundant. This redundancy is accepted because, at the hardware level, the simplicity of communication and saving of extra communications lines appears to outweigh the use of extra storage space. Figure 4-1 shows the Stage 1 to Stage 2 transformation paradigm used for for-loops.

```
        STAGE 1            |     STAGE 2
                           |
                           | -- Declaration part
                           | package Parameter is new Loop_Counter;
                           |                  -- Instantiation.
                           |                .
                           |                .
                           |                .
                           | -- Body part
                           | <<State_X>>   Parameter.Load (A, B);
                           |                  -- Load loop values.
                           |                  -- A is initial value.
                           |                  -- B is upper limit.
                           | <<State_Y>>   if Parameter.Test()  then
                           |                  -- Test the parameter
                           |                  -- versus upper bound.
  for parameter in A..B    |                 goto State_Y+1;
                           |                  -- Go to the sequence
                           |                  -- of statements.
  loop                     |               else
    Statement_1;           |                 goto State_Z+1;
                           |                  -- Exit from loop.
                           |               end if;
    Statement_2;           | <<State_Y+1>> Statement_1;
         .                 |
         .                 | <<State_Y+2>> Statement_2;
    Statement_N;           |                 .
  end loop;                |
                           |                 .
                           | <<State_Y+N>> Statement_N;
                           |
                           | <<State_Z>>   Parameter.Increment();
                           |               goto State_Y;
                           |                  -- Go back to the test.
                           | <<State_Z+1>>
                           |                  -- Continue with the
                           |                  -- rest of the program.
```

Figure 4-1: A Paradigm For-Loop Transformation

WHILE-LOOPS While-loop transformations require the instantiation of as many building block packages as required to evaluate the while-loop condition. The Stage 2 expression of a while-loop whose condition is a simple equality test is modeled in Figure 4-2.

```
<<State_Y>>     Equals.Test(
                    first_operand, second_operand, equals_result);
                goto State_Y+1;
<<State_Y+1>>   if  equals_result  then
                  goto State_Y+2;
                else
                  goto State_Z+1;      -- Exit the loop.
                end if;

<<State_Y+2>>   Statement_1;           -- Begin loop body.
                    .
                    .
<<State_Y+N>>   Statement_N;           -- End loop body.

<<State_Z>>     goto State_Y;

<<State_Z+1>>   -- ...rest of program
```

Figure 4-2:   Stage 2 Representation of a While-Loop

5. Thoughts towards a compiler

The method just presented informally emulates a multi-pass compiler that accepts as input a Stage 1 Ada program (i.e., a "normal" program confined only by restrictions we may choose to impose on the use of Ada) and produces a Stage 2 program, which is also legal, though "stylized" Ada code. This method is "compiler-like" in the sense that it is syntax driven and in that the transformations are viewed as production rules.

The Stage 1 to Stage 2 transformation involves several passes over a program unit. Backtracking within a given pass is sometimes necessary. For instance, a pass may begin by scanning the program unit and declaring the instantiation of all generic package objects that can be determined at that time, and may end with the declaration of more package objects that have been determined to be necessary while scanning the code. The passes can be organized as follows:

- Pass 1 - Transforms the declaration part of the program unit and the simple statements. Declares and instantiates packages that correspond to formal parameters and inserts code to write the actual parameter values into these packages.

- Pass 2/Part A - Transforms compound statements, that is, loops, if statements, accept statements and blocks. (Simple statements "exposed" in this step are also transformed.) Records situations that require backtracking. Also records situations that require new packages to be instantiated.

- Pass 2/Part B - Backtracks and replaces "temporary" state markers with appropriate state numbers.

- Pass 3 - Instantiates new packages whose need has been previously recorded. Transforms expressions that involve relational operators and expressions that similarly involve an increase in the number of states.

5.1. Determining concurrency within a state

Determining which actions may take place in parallel is an important part of the methodology. Reasoning can be applied to specific cases based on the function, procedure, and procedurE specifications. However, a general rule is desirable. The following principles (constraints) are adhered to:

1. At the Stage 2 level no two operations of a given package instance may be called within a given state. This applies both to multiple calls on a single subprogram contained in a generic package instance and to single calls on different subprograms of the same package. Thus, the calls

```
    Point.Load;
    Point.Test;
```

   must be invoked in separate states, whereas

```
    Point.Load;
    Slot.Test;
or
    Point.Load;
    Slot.Load;
```

   may be initiated concurrently.

2. After receiving an appropriate "Go" signal, a module M (executing a type procedurE operation) will not recognize another "Go" signal sent from a module N until after M raises the matching "I'm done" signal. If a module N were to send such a signal, its "Go" signal will be

ignored and the action that N requests of M would never take place. Furthermore, N runs the risk of mistakenly viewing the "I'm done" signal M sends upon completion of the previous operation as intended for N and will therefore proceed in error.

3. The hardware modules developed in this report have no underlying storage resource management: they allow for only one "activation record" at a given time. Thus, overlapping invocations will result in undefined behavior.

The rule is sufficient for our purposes to ensure proper behavior but no claim is made that it is always necessary. (Note that Ada semantics permit concurrent activations of operations within a package, although such permissiveness can lead to non-deterministic behavior.) The fact that a unique module is created in hardware for every variable, every computation (e.g., addition), and every comparison, suggests that control line conflicts will be avoided as long as no module is presented with more than one command at a time.

6. Stage 3: Protocol-definition Ada program

An Ada task defines a distinct thread of control. Ordinary subprogram calls by a task T are regarded as traversals along this thread of control. Since contention for subprogram activation has been eliminated by the constraints we have imposed, Go/I'm done protocols can be used safely in such cases. Inter-task communication is more complex since two separate threads of control are involved and since contention is possible. Such communication is, therefore, implemented with a four-cycle Request/Acknowledge protocol. Implementation details for both kinds of communication are introduced in the transformation from Stage 2 to Stage 3.

6.1. Motivation for Stage 3

Like its predecessor, the Protocol-definition stage is specified in legal Ada code. The discipline introduced in Section 3 is extended. The Protocol-definition stage realizes two goals:

1. New states are inserted and "Line" packages are instantiated to

specify protocols for communication between the program units expressed in the Stage 1 code.

Note that the transformations presented thus far have been concerned with communications within a given Stage 1 program unit. Since each of the original program units maps into a unique state machine/data path pair (engine), task entry calls, procedure calls, and function calls between these units cannot be represented by simple control line assertions. Instead, such communication must be implemented either using Request/Acknowledge or Go/I'm Done protocols.

2. State label numbers are converted to binary numbers, primarily to facilitate the encoding of the Stage 3 body part as an SLA state machine, which takes place in Stage 4.

In the transformation to Stage 3, the list of declared hardware modules is completed and the state machine is reduced to a sequence of if-statements, goto statements, and subprogram calls representing control line assertions.

6.2. Implementing Inter-Program Unit Communications Protocols

Stage 3 inserts protocols only for those program units that are originally specified in Stage 1. Protocols are already defined (in Stage 2) for program units that are introduced as a result of building block generic package instantiations.

In hardware representation each inter-engine communication requires two communications lines. Each line (i.e., wire) is realized by the instantiation of the generic package named "Line". The specification part for Line is:

```
generic
package Line is
  procedure Lift;
    -- Function:
    --   Assigns the logical value 1.
  procedure Lower;
    -- Function:
    --   Assigns the logical value 0.
  function Test return boolean;
    -- Function:
    --   Returns true if wire has logical value 1,
    --   else returns false.
end Line;
```

An instance of this package corresponds to a physical line whose level may be lowered, raised, or tested.

6.2.1. Transforming Procedure and Function Calls

A procedure or function X is mapped from Stage 2 to Stage 3 as follows:

1. Line packages X.Go and X.Done are instantiated.
2. The decision "if X_Go.Test()" is inserted as the initial state. (The machine remains in this state until X_Go.Test becomes true. Lines are always initialized to the logical value 0, regarded here as false.)
3. "X_Done.Lift" is made the action of the final state. The state machine of X takes the necessary actions to allow the caller to "see" the return values at the same time X_Done is sensed true.

Program units that contain procedure and function calls to other program units must also be transformed to reflect the calling protocol. For example, the action:

```
<<State_1>>  X(some_arguments);     -- Call on X
             goto State_2;
```

is transformed into:

```
<<State_1>>  X_Go.Lift;
             X(some_arguments);   -- The original action.
             goto State_2;

<<State_2>>  if  X_Done.Test  then
               -- Load the out parameters/function result
               -- into proper register(s).
               goto State_3;
             else
               goto State_2;
             end if;
```

Notice that the original invocation of X is left in the code.

6.2.2. Transforming Task Entry Calls and Accept Statements

The transformation of tasks is similar to that for subprograms. The scheme outlined in the previous subsection is followed, although "X_Req" is substituted for "X_Go" and "X_Ack" is substituted for "X_Done". Additionally, a Line package is instantiated for each entry statement of the task. This Line and the X_Req Line are "raised" concurrently by the calling task (via a calls to the respective Lift procedures). Each accept alternative in the receiving task tests the tasks request line and the corresponding entry statement line before performing the desired operation. As an example, consider the task named "Storage" that models a Read/Write memory. Storage is specified in Stage 1 as:

```
task Storage is
  entry Read(
    address: integer;
    value:   out integer);

  entry Write(
    address:  integer;
    value:    integer);
end Storage;
```

The instantiations

```
package Storage_Req   is new Line;
package Storage_Ack   is new Line;
package Storage_Read  is new Line;
package Storage_Write is new Line;
```

must be visible to Storage and all tasks which can call it.

The body of Storage is realized as:

```
<<State_0000>>  if  Storage_Read.Test() and
                       Storage_Req.Test()  then
                  goto State_0001;
                elsif Storage_Write.Test() and
                         Storage_Req.Test()  then
                  goto State_0100;
                end if;

<<State_0001>>  accept Read(
                    address: integer;
                    value:   out integer)
                do
                    -- Perform read operation.
                    -- This may take several steps
                    -- in the general case but here
                    -- we simplify to one step.
                end Read;
                goto State_0010;

<<State_0010>>  Storage_Read.Lower();
                goto State_0110;

                  .
                  .

<<State_0100>>  accept Write(
                    address: integer;
                    value:   integer);
                do
                    -- Perform write operation .
                    --
                end Write;
                goto State_0101;

<<State_0101>>  Storage_Write.Lower();
                goto State_0110;


<<State_0110>>  Storage_Ack.Lift();
                    -- Raise the acknowledge line.
                goto State_0111;

<<State_0111>>  if  Storage_Req.Test()  then
                    -- Keep Ack high until Req is lowered.
                  Storage_Ack.Lift();
                  goto State_0111;
                else
                  Storage_Ack.Lower();
                  goto State_<some_next_state>;
                end if;
                  .
                  .
```

A Stage 1 call on the Storage write operation such as

```
<<State_4>>     Storage.Write(
                    1,
                    Some_Value.Read());
                goto State_5;
```

is realized in Stage 3 as:

```
<<State_1000>>  Storage_Req.Lift();       -- Raise request line.
                Storage_Write.Lift();     -- Raise write accept line.
                Storage.Write(
                    1,
                    Some_Value.Read());
                goto State_1001;

<<State_1001>>  if  Storage_Ack.Test()  then
                  Storage_Req.Lower();    -- Test acknowledge line.
                  goto State_<some_next_state>;
                else
                  Storage_Req.Lift();
                  goto State_1001;
                end if;
```

Note that the effects of these transformations are to:

1. Force tasks to follow standard Request/Acknowledge protocol.

2. Create an implicit case statement which directs the proper accept alternative choice (e.g., State_0000 above).

6.3. Transformation to Binary Numbers

In Stage 4, states are encoded as a series of "0" and "1" cells that are connected to SR flip-flops. For example, <<State_0110>> is realized by placing "0", "1", "1", and "0" cells in the same row (AND plane) in adjoining columns a matrix called and SLA. The level associated with this row is "raised" whenever that sequence of values 0110 is stored collectively in the flip-flops. We regard raising this row's level as equivalent to being in State 0110.

To facilitate this encoding, state label numbers are transformed to binary representations as the last action of Stage 3. With the completion of the state expansions outlined earlier in this section, the state machine is fully specified.

In summary, Stage 2 to Stage 3 transformations can be performed in two passes. The first pass inserts the necessary state and package instantiations to specify the communications protocols. The second pass converts the state label numbers to binary numbers.

7. Stage 4: SLA Program

This section discusses SLA programs and their derivation from Stage 3.

7.1. Background and Use of SLA Programs

SLA is an acronym for Storage Logic Array. SLA methodology lends itself to the realization of interacting state machine/environment pairs; they are used to describe both the state machine and the data path components. The SLA concept was originally conceived by S. Patil [15] [14], extended by Patil and Welch [12] [13], and further extended by K. Smith [18]. Simply put, SLAs are "folded" Programmable Logic Arrays (PLAs) in which column and row breaks in both the AND and OR planes allow the design of independent arrays in the same circuit. "Programming" an SLA involves the placement of symbolic elements (with the help of an editor) in a manner that may result in representing an arbitrary number of independent finite state machines whose interconnection is specified by the SLA program. These symbolic elements may then be automatically translated into IC layout masks in the appropriate circuit technology. The translation of the SLA program into an integrated circuit can be viewed as the actual placement of finite SLA machines onto the active area of the chip. SLA programs make it easy for the designer to visualize the physical layout of the circuit from its logical description. A designer who thinks primarily in terms of the functional description effectively specifies the physical layout as well. Smith and co-workers have designed SLAs in I^2L, NMOS, and CMOS technologies [18]. More recent work by Smith's group has extended the SLAs based on a new concept for cell set design. The new circuits, called PPLs, are being primarily applied in the design of asynchronous state machines [4].

Our method uses SLAs in two ways:

1. The SLA modules previously developed are treated as hardware components that replace the Stage 3 generic packages. Note that no formal method is employed for the design of the SLA modules. However, each module has been simulated independently to test its correctness.

2. The state machines, including control and feedback lines, are encoded as SLAs [13].

We use SLA cells to build a library of composite "macros", which are the Stage 4 modules described in Section 3. These modules comprise the data path and are inserted using a cell substitution approach. In this sense our use of SLAs is similar to the use of macro cells [3] and Associative Logic [7].

The particular cell set employed in this work was the 5 micron NMOS set described in [17]. An SLA editor (SCLED [20]) and a SLA simulator (NSIM [19]) were built and tested at Utah; both were used extensively in this study.

7.2. Encoding of State Machines

The Stage 3 specification of a state, say, State 0110, results in the connection of the appropriate SLA cells such that the row corresponding to State 0110 goes high at the proper time. Further, in each state the levels on columns "connected" to the row of a given state are raised when the SLA is in that state. These columns are the sources of the control lines, which correspond to the operations to be initiated in that state. A two-pass method is employed to accomplish the desired encoding. This technique is presented by referring to a simple example. Consider the Stage 1 if-statement construct:

```
if  A = B  then
  C := C + 1;
else
  A := B + 1;
end if;
```

With the assumptions that "A" maps into a Register while "B" and "C" map into Counters, this construct could be specified in Stage 3 as:

```
<<State_0000>> Equals.Go(A.Read, B.Lookup, equals_result);
               goto State_0001;

<<State_0001>> if  equals_result  then
                 goto State_0010;
               else
                 goto State_0011;
               end if;

<<State_0010>> C.Increment;
               goto State_0110;

<<State_0011>> B.Increment;
               goto State_0100;

<<State_0100>> A.Write(B.Lookup);
               goto State_0101;

<<State_0101>> B.Decrement;
               goto State_0110;

<<State_0110>> null;
```

In the first pass, the states of Stage 3 are scanned sequentially. Every function and procedure call on a generic package instantiation in Stage 3 is transformed into the raising of a control line when the row corresponding to the given state "goes high". If-statements are transformed into two rows, one for each possible result of the if. The state machine layout rules employed are:

1. For simplicity, columns representing test inputs and control line outputs that are used to communicate with other state machines (program units) are placed on the left of the state machine and those that communicate to local modules are placed on the right.

2. Rows and columns are annexed as needed as the Stage 3 states are scanned. When a new Stage 3 subprogram call is discovered, a column is designated to carry the corresponding control line.

Figure 7-1 presents the result of the initial encoding pass over the Stage 3 code presented above.

```
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1
    1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6
 1: F   F   F   F   O B O B O B O B;
 2:                                ;
 3:                                ;
 4:                                ;
 5:                                ;
 6:                                ;
 7: 0   0   0   0 S   + + +        ;
 8: 0   0   0 S 1 R 0              ;
 9: 0   0   0 S 1   1              ;
10: 0   0 S 1   0           +      ;
11: 0   0 S 1 R 1 R           +    ;
12: 0   1   0   0 S       +     +  ;
13: 0   1   0 S 1 R               +;
14: 0   1   1   0                  ;
                    ^ | | | | | | |--> B.Decrement
                    | | | | | | |----> A.Write
                    | | | | | |------> B.Increment
                    | | | | |--------> C.Increment
                    | | | |----------> B.Lookup
                    | | |------------> A.Read
                    | |--------------> Equals.Go
                    |-------- result from Equals
```

Figure 7-1: First Pass Stage 4 Encoding

Note how state 0000 (row 7) raises columns 10, 11, and 12. This row corresponds to the "Equals.Go(A.Read, B.Lookup,...)" operations specified for state 0000 in the Stage 3 code above. State 0001 (rows 8 and 9) corresponds to the if-statement. Row 8 "goes high" if the result from the comparator carried in column 9 is false (i.e. a /= b). Row 9 goes high if the result is true (a = b). Note how new columns are added on the right as new procedure and function calls are scanned in the Stage 3 code. Note also how the B.Lookup (column 12) is raised in State 0000 (row 7) and in State 0100 (row 12). The second time "B.Lookup" is scanned in the Stage 3 code we remember that a column was already dedicated to this control line; we don't dedicate another. Since this simple circuit does not communicate with other state machines, all control line firings are on the right side.

In the first pass the "+", "1", and "0" cells are placed only as the need for them is discovered. A dispersed layout often results. The second manual pass re-arranges the control lines to group lines that are directed to the same module. Thus, the second pass merely clusters the control lines, arranging them according to their destination. The effect of the second pass is to simplify routing of the control lines to the modules. Figure 7-2 presents the result of re-arranging of the columns of Figure 7-1. Note how commands going to the same module are now on adjacent columns.

```
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1
    1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6
 1: F   F   F   F   O B O B O B O B;
 2:                                ;
 3:                                ;
 4:                                ;
 5:                                ;
 6:                                ;
 7: 0   0   0   0 S   + +   +      ;
 8: 0   0   0 S 1 R 0              ;
 9: 0   0   0 S 1   1              ;
10: 0   0 S 1   0                 +;
11: 0   0 S 1 R 1 R           +    ;
12: 0   1   0   0 S       + +      ;
13: 0   1   0 S 1 R             +  ;
14: 0   1   1   0                  ;
                    ^ | | | | | | |--> C.Increment
                    | | | | | | |----> B.Decrement
                    | | | | | |------> B.Increment
                    | | | | |--------> B.Lookup
                    | | | |----------> A.Write
                    | | |------------> A.Read
                    | |--------------> Equals.Go
                    |-------- result from Equals
```

Figure 7-2: Second Pass Stage 4 Encoding

7.3. Layout, Routing and Busing Issues

An algorithmic method for cell layout and routing has not yet been incorporated into our method. Reference [6] discusses a simple manual routing method that utilizes the fact that the declaration part of a given Stage 3 program unit specifies the modules utilized by that unit.

As mentioned earlier, engines that are physical representations of tasks communicate through the use of the Request/Acknowledge protocol. In the hardware realm, such engines communicate via buses. A circuit derived by our method may include several buses, which may be private (non-contention) or public (with potential for contention between the users). Both types support the Request/Acknowledge protocol. It is well-known that a Request/Acknowledge protocol strategy will not work on a contention bus without some sort of arbitration mechanism. The Request/Acknowledge protocol implemented here closely follows the scheme outlined by Seitz [16], and appears to be adaptable to his arbitration scheme. Bus issues are detailed further in [6].

8. Conclusions

The transformation methodology described in the preceeding sections was developed and exercised in conjunction with an extensive and non-trivial case study [6]. The algorithm developed for that exercise is a possible model for the behavior of the Ada selective wait statement, itself initially specified as an Ada program consisting of a set of intercommunicating Ada server and requestor tasks. The transformation rules were only applied to a subset of the program. Application of the rules resulted in two SLA programs whose behavior was tested with the simulator NSIM.

The case study [6] provided a "real" example of rule-based transformations which covers the significant portion of the Ada-to-Silicon "spectrum". No theoretical stumbling blocks were encountered in this process, which suggests that there is nothing in principle to invalidate the concept that such transformations may be automated. On the other hand, we have not yet formalized these transformation rules as concrete algorithms. There is the additional challenge of reaching practical and competitive circuits with this approach.

We have experimented the intriguing concept of using Ada itself as an intermediate language in the mapping process. For this purpose we have found important ways to exploit Ada's abstraction features:

1. In mapping Ada program variables to instantiations of generic packages to pre-defined IC modules.

2. In mapping Ada subprogram and task calls to specific hardware protocols.

The end result of successful research in this area can be that the traditional hardware logic design activity will become increasingly a programming activity that is keyed to the use of high-order programming languages for system specification. Such an evolution will progress, however, only as rapidly as we succeed in evolving a new class of high-quality compilers for hardware.

REFERENCES

1. Aho, A., and Ullman, J., Principles of Compiler Design, Addison-Wesley, Reading, Mass., 1977.

2. Barnes, J., Programming in Ada, Addison-Wesley Publishers Ltd., 53 Bedford Square, London, WC1B 3DZ, International Computer Science Series, 1982.

3. Carey, J. and Blood, B., "Macrocell Arrays-An Alternative to Custom LSI," Proceedings Semi-Custom Integrated Circuit Technology Symposium, Institute for Defense Analysis, Science and Technology Division, May 1981, pp. 19-37.

4. Carter, T., "ASSASSIN: An Assembly, Specification and Analysis System for Speed-Independent Control Unit Design in Integrated Circuits Using Path Programmable Logic (PPL)," Master's thesis, University of Utah Computer Science Dept., June 1982.

5. U.S. Department of Defense, Military Standard ADA Programming Language, U.S. Department Of Defense, Washington D.C., 1980.

6. Drenan, L., "On Transforming Ada to Silicon," Master's thesis, University of Utah Computer Science Dept., August 1982.

7. Greer, D., "An Associative Logic Matrix," IEEE Journal of Solid State Circuits, Vol. SC-11, October 1976, pp. 679-691.

8. Organick, E., "Programmer's Introduction to Hardware Design", unpublished course notes used at the University of Utah

9. Organick, E., " Semiannual Technical Report: Transformation of Ada Programs Into Silicon," Tech. report UTEC-82-020, University of Utah Computer Science Dept., March 1982, DARPA Order No. 4305.

10. Organick, E.; Boll, S.; Davis, A.; Griss, M.; Hayes, A.; Hollar, L.; Huber, R.; Lindstrom, G.; Rushforth, C.; Smith, K.; and Subrahmanyam, P., "Transformations of Ada Programs into Silicon : A Research Proposal to Defense Advanced Research Projects Agency," University of Utah, March 1981.

11. Organick, E., and Lindstrom, G., "Mapping High-Order Language Units Into VLSI Structures," Proc. COMPCON 82, IEEE, Feb. 1982, pp. 15-18.

12. Patil, S. and Welch, T., "A Programmable Logic Approach for VLSI," IEEETrans, Vol. C-28, Sept 1979, pp. 594-601.

13. Patil, S., "On Testability of Digital Systems Designed with Storage/Logic Arrays," IEEE International Conference on Circuits and Computers, 1980, IEEE, New York, 1980.

14. Patil, S., "Micro-control for Parallel Asynchronous Computers," 1975 Proceedings Euromicro, Euromicro, 1975, North-Holland Publishing Company.

15. Patil, S., "An Asynchronous Logic Array," Tech. report TM-62, MIT, May 1975, Project Mac.

16. Seitz, C., "Ideas About Arbiters ," LAMBDA, Vol. 1, No. 1, First Quarter 1980, pp. 10-14.

17. Smith, K., "Design of Integrated Circuits with Structured Logic Using the Storage Logic Array (SLA) Definition and Implemantation," PhD dissertation, University of Utah, March 1982.

18. Smith, K.; Carter, T.; and Carter, T., "Structured Logic Design of Integrated Circuits Using the Storage/Logic Array (SLA)," IEEE Transactions on Electron Devices, Vol. ED-29, No. 4, April 1982, pp. 765-776.

19. Nelson, B., NSIM User's Manual: University of Utah VLSI Research Group, 1981.

20. Nelson, B., SLED User's Manual: University of Utah VLSI Research Group, 1981.

Ada Specifications for the DoD Internet Protocol:

The INM _ OUT Submodule,

Report No. 1

by

Gary Lindstrom, Elliott I. Organick,
Daniel Klass, and Michael P. Maloney

Department of Computer Science

Univerisity of Utah

Salt Lake City, Utah 84112

November, 1982

This work was sponsored in part by the Defense Advanced Research Project Agency (DARPA) under contract number MDA 903-81-C-0414.

Table of Contents

List of Figures

Abstract

This describes the status of the Internet Protocol (IP) example being pursued as a case study by the Utah Ada to Silicon Project. This document provides three contributions: (1) A general introduction to the Internet Protocol for those unfamiliar with it, (2) A discussion and "road map" through the structure of the Ada code that specifies the submodule representing IP, which we have named INM_OUT, and (3) A complete listing of the source Ada code for INM_OUT that is being used to guide the transformation of this submodule into silicon. Parts 1 and 2 summarize the function of the IP and our major design decisions.

Other references [2, 3, 4] also include discussions of the IP case study and our approach to mapping the IP into silicon. The source listings in part 3 have been compiled using the Intel 432 Ada compiler version available to us at this time. We have coded the complete INM_OUT submodule in Ada and have succeeded in compiling most of it for execution on the Intel iAPX 432 system except for statements and declarations associated with uses of the Ada rendezvous construct.

[As later versions of the Intel compiler become available, we expect not only to be able to compile the full module using rendezvous syntax and semantics, but to execute it in this mode as well. In the meantime we are working with a version of the code, not given in this report, that simulates each rendezvous via Send/Receive primitives instantiated through use of the Ada generic package mechanism.]

1. What is the Internet Protocol?

The IP is one level in a hierarchy of protocols designed to provide a uniform means of transmitting messages from one computer system ("Host") to another, over an interconnected system of networks (or "nets"). We term each of the individual networks a "local net", even though such a net could be worldwide in scale, as is the Arpanet. The overall assemblage of these networks is called the internetwork, or "Catenet". Hosts directly interfacing to two or more local nets are called "gateways".

The primary reference for the IP is [5]. Quotations appearing in this document without explicit attribution are taken from this reference.

1.1. Protocol Hierarchies

It is important to understand the IP's position in the protocol hierarchy, as well as what it means at all to speak of a protocol hierarchy. Fig. 1-1 depicts this layering, where TCP stands for *Transmission Control Protocol* [6], the most common protocol above the IP, and LNP stands for *local net protocol*, which we leave unspecified here.

```
                .
                .
                .
 +------------------------+
 |      higher-level      |
 +------------------------+
 |          TCP           |
 +------------------------+
 |          IP            |
 +------------------------+
 |          LNP           |
 +------------------------+
 |     line protocol      |
 +------------------------+
                .
                .
                .
```

Figure 1-1: Protocol layering.

Depending on one's perspective, there are several ways of looking at protocol layering:

1. *Modularity and intermediate languages*: One may view the hierarchy as a conceptual (or real) framework for organizing an implementation into *modules* each performing a *language mapping* function. For example, we will use the terms Transmission Control Module (TCM), Internet Protocol Module (INM), and Local Net Module (LNM), to refer to the separate modules (actual or conceptual) implementing the TCP, IP, and local net protocol, respectively.
2. *Levels of abstraction*: This viewpoint, in many ways the most fundamental, regards each protocol as specifying a level of communication abstraction. That is, at each level certain aspects of the overall internetwork communication problem are solved, and rendered invisible to higher level protocols. For example, we shall see that the IP deals with local net packet sizes, so the TCP can function under the abstraction of essentially unlimited packet sizes.
3. *Division of duties*: Correspondingly, the modules at each level accomplish certain duties in support of the particular level of abstraction they implement. For example, we shall see that an INM accomplishes its packet size abstraction by implementing a message fragmentation and reassembly process.
4. *Nested enveloping*: Still another viewpoint, which we develop further in section 2.2, focuses on the "wrapping" of additional layers of control information ("headers") on messages as they descend in the protocol hierarchy, and the corresponding "upwrapping" as they rise. At each level, the sending and

receiving modules communicate through parameters packed in the headers of data passed to the next level.

1.2. The Role of the IP

The IP fundamentally provides a means of transmitting uninterpreted messages (*segments*) between Hosts on possibly different local nets. The INMs accomplish this transmission by packaging these segments in special data blocks termed *datagrams*, for transmission via one or more local nets.

In performing its part of this internetwork service, the IP is concerned with two principal duties:

1. *Internet addressing*: picking the desired "next hop" gateway for nonlocal messages. and
2. *Fragmentation* and *reassembly*: splitting and merging messages that cannot be transmitted intact due to inadequate local net packet sizes.

These duties can be explained metaphorically as follows. The IP functions like a department–to–department mail service within an industrial organization. Each department has a mail room, which deals with one or more courier services. When someone in a source department has an item to send to another department, he or she wraps it in an unmarked folder and deposits it in an out basket of the local mail room, with a delivery slip attached giving instructions.

The mail room prepares the folder for transmittal by inserting it into a company mail envelope, with the delivery instructions written on its exterior. It then selects a courier serving the destination department's mail room, and gives the envelope to the service's agent. The agent then puts the company mail envelope into one of the service's own standard envelopes, and enters it into its shipping system. At the destination the process is reversed: the courier agent strips off the courier service envelope and delivers it to the mail room, which in turn recreates a delivery slip from the instructions on the company mail envelope, strips of the company mail envelope and, puts the folder (with delivery slip attached) into one of the department's in baskets. The in basket is selected according standing processing instructions, based on the contents of delivery slips.

However, two complications may arise in accomplishing this folder transmittal:

1. The courier services available to the source mail room may not directly service the destination department. In this case, the mail room determines a (remote) courier service directly serving the destination, and looks the service's name up in a routing table. This table gives the name of a department whose mail room has agreed to transfer mail to the destination department, as well as the name of a courier directly serving the transfer department. The source mail room then gives its company mail envelope to the shared courier service, which conveys it to the transfer department's mail room. The envelope is then relayed out via another courier service, which the transfer mail room determines according to its own routing table.
2. The second difficulty may be that the given folder size exceeds the capacity of largest envelope available from the selected courier service. In this case, the mail room takes the liberty of partitioning the folder's contents so that each portion will fit into a service envelope. However, before passing each portion to the courier agent, it marks on the portion's company mail envelope that portion's sequential position in the original folder. This permits the portions to be reassembled into one folder in the destination mail room.

This thinly disguised analogy maps into the IP world as follows:

—A *department* is a Host, and a *courier service* is a local net.

—A *mail room* is an INM, and each *courier agent* is an LNM.

—A *folder* is a data segment for transmission over the catenet.

- An *out basket* is a **SEND** call, and an *in basket* is a **RECV** call. **Delivery slips** are **SEND/RECV** call parameters.
- Each piece of *company mail* is a datagram if it contains a complete segment, and a (datagram) fragment otherwise. (For convenience, we consider unfragmented datagrams to be "fragments" as well.)
- *Transfer mail rooms* are gateways.
- Finally, a *courier mail envelope* is of course a local net packet.

(End of postal terminology, and resumption of Postel terminology.)

1.3. The TCM INM Relationship

The manner by which the TCM communicates with the INM is not standardized. However, the IP manual [5] illustrates one possible implementation through a pair of procedure calls **SEND** and **RECV**.

The sending TCM issues an INM call of the form

SEND(src, dst, ..., BufPTR, len, ...)

when it wishes to send a segment to a destination Host. Parameters src and dst give the Internet addresses of the source Host (presumably itself) and destination Host, respectively. Internet addresses are simply the concatenation of a net number and a Host number. The segment to be transmitted is of length len (in 8-bit bytes, or "octets"), and may be found in memory location BufPTR. (Omitted parameters will be discussed in section 2.1.)

If all goes well, this segment will be presented in due course to the TCM at the destination Host. It takes delivery of the incoming segment by completing a mating **RECV** call on its INM, which we assume was awaiting its arrival:

RECV(BufPTR, ..., src, dst, ..., len, ...),

where **src**, **dst**, and **len** are value-returning ("OUT") parameters, and **BufPTR** provides a pointer to a preallocated segment buffer in the receiving TCM. Although **dst** is an OUT parameter, we may assume that all segments delivered will have **dst** equal to the Host's Internet address. Note that all through traffic at a gateway is handled by its INM without involvement with the Host's higher level protocols (i.e. without TCM **SEND/RECV** handling).

The TCM, for its part, implements several higher-level aspects of the internet communication process:

- reliability (e.g. acknowledgements and retransmissions);
- error control at the segment level (i.e. checksumming TCP headers, etc.);
- flow control (controlling the rate at which segments are delivered to the INM);
- multiplexing (management of multi-purpose segments);
- connections (reserved portions of transmission capacity), and
- precedence and security (managing degrees of urgency and confidentiality of segments).

1.4. The INM LNM Relationship

The interface between the INM and LNM is not specified in [5]. One may speculate, however, that it could follow the general form of the **SEND/RECV** calls at the TCM-INM interface.

That is, when an INM has a fragment to send out on a local net, it issues a **SEND** call in the net's LNM as follows:

SEND(src_ln, dst_ln, ..., FBufPTR, Flen)

Parameters src_ln and dst_ln give the numbers of the sending and target Hosts on this net. Recall dst_ln will designate either this fragment's Internet destination Host, or the Host serving as its next gateway. FBufPTR and Flen indicate the memory location and extent of the fragment constructed by the INM.

Delivery of local net packets by LNMs at target Hosts is accomplished by completion of an INM call (which again we assume is waiting) of the form:

RECV(FBufPTR, ..., src_ln, dst_ln, Flen),

where src_ln, dst_ln, and Flen are OUT parameters serving the obvious functions

It is useful to note the communication functions provided by LNMs:

—packet formation and transmission;

—local net status control;

—routing of packets within each local net.

2. A Closer Look at IP Functionality

2.1. TCM Interface

The full parameterization of the **SEND/RECV** calls at the TCM–INM interface is as follows:

SEND(src, dst, prot, TOS, TTL, BufPTR, len, Id, DF, opt, OUT result)

—**src, dst**: Internet source and destination addresses.

—**prot**: the next level protocol in effect (e.g. at the TCM level). Several of these have already been assigned (see [7]); TCP, for instance, has assigned number 6

—**TOS**: type of service (normal, high throughput, etc.) requested by the TCM.

—**TTL**: time to live, a time (in seconds) after which the datagram derived from this segment can "self–destruct" if not delivered (see section 2.5).

—**BufPTR, len**: TCM segment pointers.

—**Id**: segment identification tag, for reassembling fragments derived from this segment (see section 2.5).

—**DF**: a "don't fragment" switch.

—**opt**: options to be observed in transmitting the segment (see section 2.6).

—**result**: an OUT parameter in {OK, error}; OK = "datagram sent ok"; error = "error in arguments, or local network error".

The corresponding **RECV** call issued by the TCM at the destination Host has a similar parameterization:

**RECV(BufPTR, prot,
OUT result, OUT src, OUT dst, OUT TOS, OUT len, OUT opt)**

The purpose of these parameters should be evident from consideration of the corresponding **SEND** parameters. Note, however, that two are IN (read–only):

—**BufPTR**: a pointer to buffer preallocated by the TCM for receipt of the incoming segment.

—**prot**: an indication of which higher level protocol version this **RECV** call can accommodate.

2.2. Datagram Formatting

As mentioned in section 1.1, a fruitful way of looking at protocol layering is to consider the levels of envelope nesting that surrounds the raw data transmitted. This is illustrated in fig. 2-1.

```
                                    .
                                    .
                                    .
                     +--------------------+   ------------------------------
LNM parameters       |  local net header  |                                ^
                     +--------------------+   ----------------------    packet
INM parameters       |     IP header      |                         ^
                     +--------------------+   -----------   fragment
TCM parameters       |    TCP header      |       ^
                     +--------------------+    segment
                     |    data buffer     |       v            v         v
                     +--------------------+   ------------------------------
                                    .
                                    .
                                    .
```

Figure 2-1: Data enveloping.

Since we are concerned primarily with the IP level, it is useful to look in more detail at the format of an IP fragment (see fig. 2-2).

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  ---------
|Ver= 4 |IHL= 8 |Type of Service|      Total Length = 576       |      ^
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Identification = 111     |Flg=0|   Fragment Offset = 0     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  fixed
|  Time = 123   | Protocol = 6  |        Header checksum        |    layout
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        source address                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     destination address                       |      v
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  ---------
| Opt. Code = x | Opt. Len.= 3  | option value  | Opt. Code = x |      ^
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  full
| Opt. Len. = 4 |          option value         | Opt. Code = 1 |    words
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Opt. Code = y | Opt. Len.= 3  | option value  | Opt. Code = 0 |      v
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  ---------
|                             data                              |      ^
\                                                               \
\                                                               \  one or more
|                             data                              |    octets
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |      v
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+  ---------
```

Many of these fields are directly transferred from corresponding **SEND** parameters. However, a few bear clarification:

—**Ver**: version of the IP header layout.

—**IHL**: total header length, in multiples of 4 octets (32 bit words).

—**Type of service**: a one-octet encoding of the type of service which the datagram should be given en route to its destination. (This encoding is apt to be mapped to other representations as the datagram moves first to the local net level and then to other networks en route to the destinatin network.)

—**Total length**: total length of the datagram, in octets.

—**Flg**: three bits $b_0b_1b_2$, where b_0 must be zero, b_1= 1 iff the datagram should not be fragmented, and b_2= 1 iff this fragment is not the final one of its datagram.

—**Fragment Offset**: gives the position of this fragment's message data within its original segment, in units of 8 octets (64 bits). The first fragment of a datagram has offset zero.

—**Header checksum**: from [5], p. 14:

"The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header. For purposes of computing the checksum, the value of the checksum field is zero."

2.3. The Internet Addressing Function

Internet addresses actually have three formats, providing for a few nets with relatively many Hosts, and many nets with relatively few Hosts. These formats are:

—*Class A*: a lead 0, followed by a 7-bit net name, followed by a 21-bit Host name.

—*Class B*: a lead 10, followed by a 14-bit net name, followed by a 16-bit Host name.

—*Class C*: a lead 110, followed by a 21-bit net name, followed by an 8-bit Host name.

Several Class A network names have already been assigned [7].

As mentioned in section 1.2, the INM addressing function deals only with outgoing datagrams, and amounts to picking the target Host on the next local net. This will involve use of:

1. A gateway table, which will need to be updated periodically to reflect long term additions and deletions of nets to the Internet system, as well as shorter term changes in gateway availabilities.
2. Use of specific routing instructions, as given in the datagram options (see section 2.6).

2.4. Fragmentation

Fragmentation occurs on outgoing datagrams which will not fit into a single local net packet. Note that fragment headers can be constructed without examination of the data segment to be transmitted. This means that a buffer the size of a local net packet could suffice for fragmentation if space is at a premium. The IP specification [5] gives an example fragmentation procedure (p. 26).

2.5. Reassembly

The IP specification also gives an illustrative reassembly algorithm (p. 28). The key points from our perspective are the following:

—Reassembly is done only at Internet destinations, and *not* at gateways or other intermediate Hosts (since we cannot be sure all fragments derived from a given datagram will follow the same routing).

—Datagram fragments are reunited on the basis of a key formed from four fields of the fragment headers: source, destination, protocol, and identification. Sending TCMs must choose identification fields such that this 4-tuple is unique throughout the Internet system for the lifetime of a datagram.

—Strangely enough, fragment headers do not include the *overall* size of a (reassembled) datagram. Hence preallocation of a complete buffer for each incoming datagram is not generally feasible, unless either a small limit is imposed on incoming datagram size, or the datagram arrival rate is assumed to be low.

—Various anomalies can occur in the arrival of fragments, e.g. duplications, reorderings, and omissions. The INM is free to handle these however it wishes, except that fragments with headers that fail the checksum test must be destroyed. Fragments are "aged" by decrementing their TTL field as they pass through the Internet system. Each INM handling a fragment charges its processing time, with a minimum of one (second) each. Presumably, the TTL for a datagram under reassembly is the minimum of the TTLs for its delivered fragments When this TTL reaches zero, the partially formed datagram is destroyed, and the buffer is released.

2.6. Options

Options indicate special handling for datagrams, as requested by the sending TCM. The *use* of options is optional, but their **implementation** is mandatory.

The essential options are summarized below, omitting "null-options" such as no-ops, padding, etc. An asterisk indicates that the option is copied in every derived fragment.

- —***Security**: for sending "security, compartmentalization, handling restrictions, and TCC (closed user group) parameters".
- —***Loose Source and Record Route (LSRR)**: for specifying a series of internet addresses through which a datagram is to be routed. The routine is loose because "the gateway or Host IP is allowed to use any route of any number of other intermediate gateways to reach the next address in the route". The route is recorded in the sense that a pointer packaged as part of the option is advanced as each intermediate address is reached.
- —***Strict Source and Record Route (SSRR)**: similar to LSRR, except that "the gateway or Host IP must send the datagram directly to the next address in the source route through only the directly connected network indicated in the next address to reach the next gateway or Host specified in the route."
- —**Record Route**: requires each INM handling the fragment to concatenate its address into the space allocated for this option (if sufficient space remains).
- —***Stream Identifier**: "provides a way for the 16-bit SATNET stream identifier to be carried through networks that do not support the stream concept."
- —**Internet Timestamp**: indicates that each INM handling the fragment should concatenate its time of receipt (in milliseconds since midnight UT) into the space allocated for this option.

2.7. Internet Control Message Protocol (ICMP)

The INM must implement special protocol that is companion to the IP for reporting errors in datagram transmission and requesting special INM services. This protocol, termed the ICMP [8], is mandated as follows:

"ICMP uses the basic support of IP as if it were a higher level protocol, however, ICMP is actually an integral part of IP, and must be implemented in every IP module."

ICMP datagrams may be recognized by INMs through the special **prot= 1** header indication. For obvious reasons, ICMP datagrams are not sent regarding errors in delivering ICMP datagrams. Briefly, their varieties are as follows:

1. **Destination unreachable**: a receiving gateway could not transfer a datagram, or a don't fragment request could not be honored.
2. **Time exceeded**: a *first* fragment, or unfragmented datagram, was superannuated.
3. **Parameter problem**: a datagram header was found to be malformed.
4. **Source quench**: a destination Host requests a slower rate of transmission from a source Host.
5. **Redirect**: a gateway advises a Host not to route traffic to a particular distant net through it.
6. **Echo or echo reply**: used to "reflect" datagrams back from destinations to sources, for testing purposes.
7. **Timestamp or timestamp reply**: similar to echo and echo reply, but with a destination timestamp.
8. **Information or information reply**: used for querying "what network is this?".

3. Current Design

We summarize here the principal features of the AtoS approach to implementing the INM, as well as remarks on the current status of that implementation.

3.1. Major Design Decisions

There have been two major design decisions thus far.

1. The first is to split the INM into three submodules: an INM_OUT dealing with traffic outbound on a given local net, an INM_IN similarly handling inbound traffic, and an INM_SRV tying them together and interfacing to the Host(s). We envision one INM_IN and INM_OUT pair for each local net interface, but only one INM_SRV per INM.
2. The second decision is to use a two-phase Ada rendezvous to implement both the upper (TCM) and lower level (LNM) interfaces. In each case, a task **call** is performed by the initiator of the data transfer action, with the receiver servicing the transfer through an appropriate entry. When the data transferred has been fully processed, a reciprocal rendezvous takes place (with **call** and **entry roles** reversed) to report the success or failure of that processing. [An alternative formulation, based on passing messages via *ports* such as is done in the i432 architecture, is also under consideration.]

Division of functional responsibilities:

1. INM_SRV:
   - a. Receive segments from and deliver segments to TCMs in the Host(s) served.
   - b. Accept incoming segments from the INM_INs, and
     - i. deliver via local Host RECV calls all segments so addressed, and
     - ii. (if implementing a gateway) route to appropriate INM_OUTs all through traffic.
   - c. Maintain a gateway transfer table, used to route all outbound segments (whether from a local Host or neighboring INM_IN). If an outbound segment has a non-local net name in its destination address, that net name is used as a key to select the appropriate next gateway directly reachable by a local net served.
   - d. Implement ICMP message generation and transfer.
   - e. Handle options:
     - i. Security: reject all classified traffic, perhaps with an ICMP report of "destination unreachable".
     - ii. LSRR, SSRR, and record route.
     - iii. Timestamping: (note this requires a time of day service, presumably from the TCM).

   [Note that all message traffic through the INM_SRV is in *segment* form; *datagram* (or fragment) form is used solely within INM_IN and INM_OUT submodules.]
2. INM_OUT:
   - a. Form fragments from segments received from INM_SRV.
   - b. Deliver fragments to the LNM_OUT of its assigned local net, along with their local net addresses (final or gateway), as provided by INM_SRV.
   - c. Map the Internet type of service parameter to an appropriate local net type of service, or reject fragment if this is not possible.
3. INM_IN:
   - a. Receive fragments from the LNM_IN of its assigned local net.

b. Reassemble fragments into complete datagrams (destination fragments only).

c. Delete overage and erroneous fragments (note this requires a timing pulse at least once each second).

4. Ada Specifications for INM_ OUT A Road Map

The INM_ OUT module, whose functionality is described in the preceding section, has been specified in full in Ada code. The purpose of this section is to review the structural organization of this code as a set of interrelated Ada packages, embedded tasks, and auxiliary procedures. The code itself is listed in the Appendix as a series of 14 separate compilation units.

4.1. Communication between INM_ OUT and its "neighbor" modules

To better understand the code organization, it is useful first to visualize the communication channels that are assumed to exist between INM_ OUT and other modules [1]. These channels suggest the important intertask communication of the Ada code to be described. Recognition of these channels determines the gross organization of the code that embodies this modular organization. Figure 4-1 shows the channels not only between INM_ OUT and its "neighbors", but also identifies two other important channels that are assumed to exist; the latter, however, are not detailed within the code to be described.

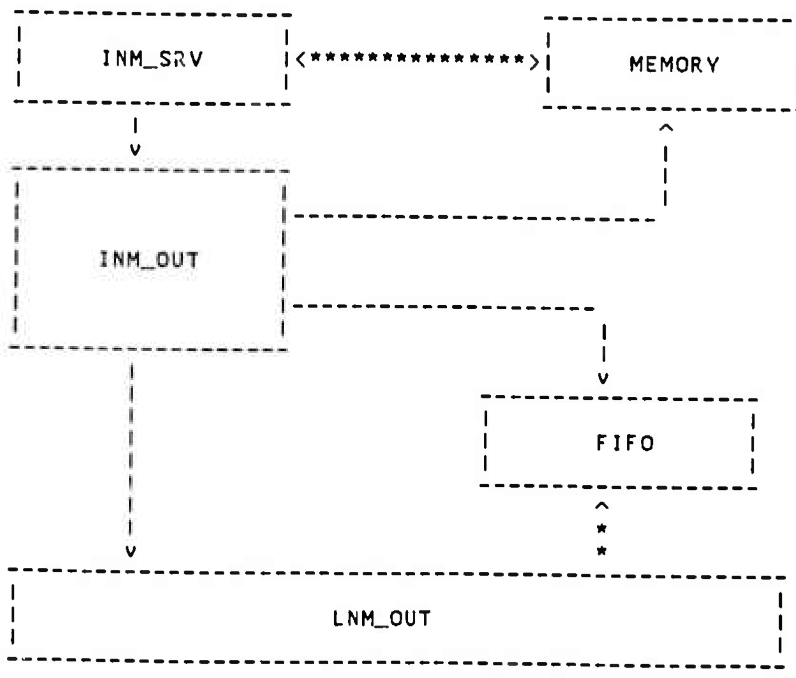

Figure 4-1: Communication channels (tasking interfaces) between INM_ OUT and its "neighbor modules". Directed arcs indicate direction of intertask requests (Ada entry calls). Arcs composed of asterisks (*) represent assumed communication channels that are not now modeled in the Ada code.

Discussion in the preceding section has already explained the role of the INM_ SRV and LNM_ OUT modules. The module marked "MEMORY" is, depending on the specific implementation, either a memory to which INM_ SRV and INM_ OUT have shared access or a control unit that governs access to some such memory unit. The module marked FIFO is assumed to be a hardware unit functioning as a first-in-first-out queue. Outbound datagram fragments are passed through the FIFO module to LNM_ OUT. The FIFO must be capable of holding at least one (maximum-sized) datagram fragment. The module is assumed to operate under asynchronous and independent control; it can be fabricated either as a separate IC or assembled from off-the-shelf ICs.

The arrowheads in Figure 4-1 indicate direction of intermodule requests, which are specifiable in Ada as task entry calls. Ada task entry calls can specify transmission of both inbound and outbound information. Completion of the rendezvous initiated by an Ada task entry call can therefore have the effect of both sending data to the callee and receiving data from the callee. Even though such a "transaction" may always be initiated by a particular party (the caller), message information on the channel can flow in one or both directions, as a result of a single call (request).

Thus, INM_OUT receives requests from INM_SRV and issues requests to LNM_OUT as well as to FIFO and to MEMORY. Depending on the nature of these requests, message information flows either to or from INM_OUT, or in both directions. These details are specified in the code itself.

Requests from INM_SRV to INM_OUT are of two kinds:

1. Messages for the purpose of providing INM_OUT with initialization information. An initialization request is a message that supplies INM_OUT with a pointer to locate and acquire, via MEMORY, the actual initialization values.
2. Messages that request transmission of datagrams. A transmission request contains a pointer which INM_OUT can use to locate and acquire, via MEMORY, the actual datagram prepared (or transshipped) by INM_SRV.

A message request from INM_OUT to MEMORY may either supply MEMORY with a pointer value or receive from MEMORY a data value.

A datagram fragment is sent by INM_OUT in the form of a message request to the FIFO unit. INM_OUT uses the channel to LNM_OUT to issue requests for confirmation that the latter has received a datagram via the FIFO. In a like manner, the channel from INM_SRV and INM_OUT is used by the former to obtain confirmation that the latter has correctly processed the preceding request.

The channels between INM_SRV and MEMORY, while important to the operation of INM_SRV, are not relevant to the current discussion.

4.2. Package and task structure of the corresponding Ada code

In the Ada code, each of the modules discussed in connection with Figure 4-1 is modelled by a package, the principal one for our purposes being the package for INM_OUT which is named Inm_Out_Module. Both the specification part and the body part of Inm_Out_Module have been coded. By contrast, it is only necessary for our purposes to supply the specification parts for the LNM_OUT, FIFO, and MEMORY modules, since only the specification parts are relevant in the design of INM_OUT. By similar reasoning, since INM_SRV issues entry calls into INM_OUT and not vice versa, it is unnecessary to consider even the specification part of INM_SRV; for this reason, there is no package representing INM_SRV in the code section displayed in this report.

4.3. Definition packages

The full Ada code for INM_OUT, in the form of an Ada package, has been deliberately composed with certain declarative information factored out; the factored information takes the form of a hierarchy of three (auxiliary) definition packages. These packages contain type information (type and subtype declarations and their corresponding representation clauses, if any) as well as constant information (constant declarations); these declarations are expected to be common to either INM_OUT or to INM_SRV, or to both when these modules are encoded as Ada packages sometime in the future. Thus, the "root" definition package is named In_Out_Srv_Defs, because the contained declarative information is common to all three parts of the Internet Module; the subsidiary package Inm_ In_ Out_ Defs contains declarative information common to both INM_IN and INM_OUT and depends on the declarative information in In_ Out_ Srv_ Defs. Finally, the definition package named Inm_ Out_ Defs contains declarative information of relevance only to INM_OUT and to the modules (LNM_OUT, MEMORY, and FIFO) to which it makes requests for service. Figure 4-2 shows the full dependency graph that has resulted from this decision to factor out common declarative information. The graph also reveals that the packages representing MEMORY, FIFO, and LNM_OUT modules have also been specified to depend on certain of the definition packages.

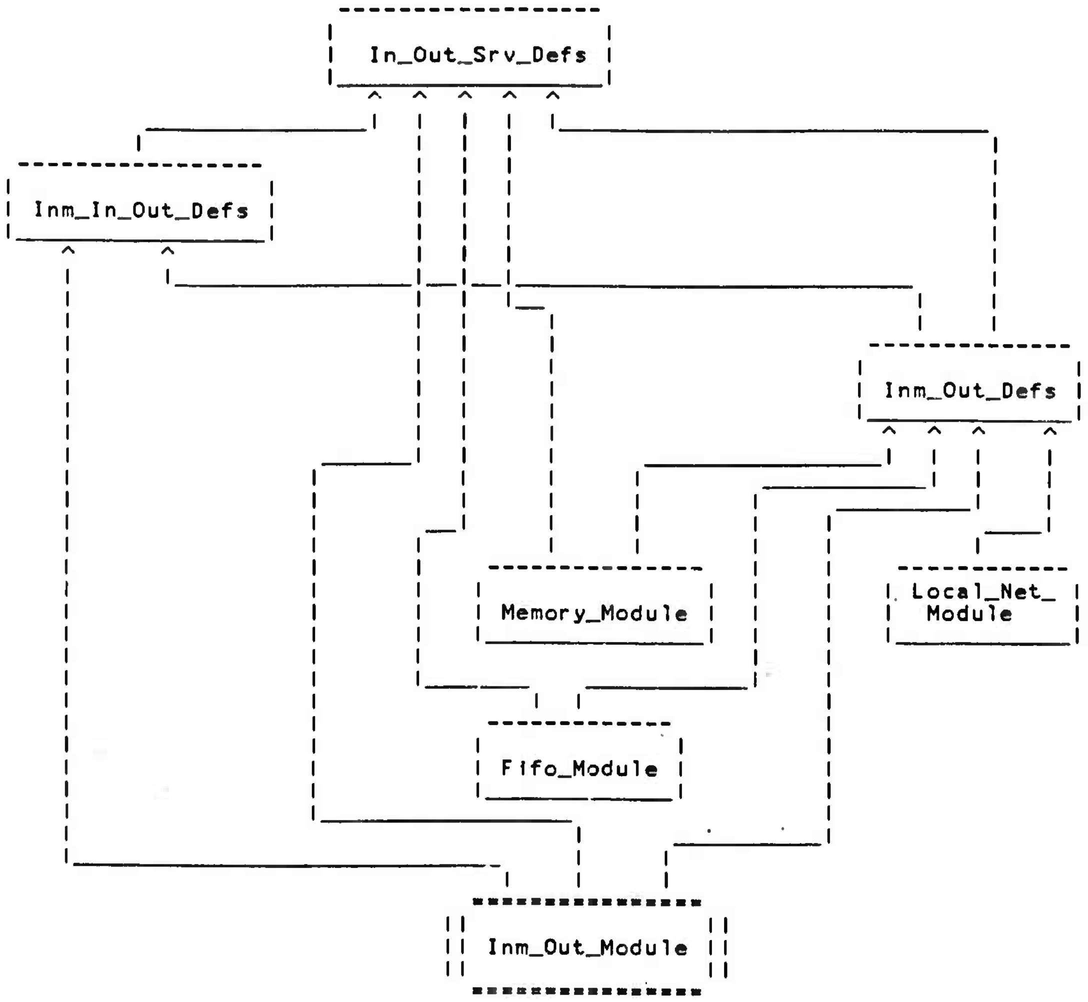

Figure 4-2: Graph illustrating the dependence of the module packages on certain auxiliary definition packages.

4.4. Tasks defined within the Inm_ Out_ Module package

Three tasks are declared within the Inm_ Out_ Module package.

1. The main task, named Inm_ Out, interfaces with INM_ SRV and with LNM_ OUT such that a pipeline effect is achieved for speeding datagrams along the outbound data path: Host module —> INM_ SRV —> INM_ OUT —> LNM_ OUT.
2. An auxiliary (server) task, named Read_ Init_ Parameters, which obtains from host-related memory the initial parameter values needed to perform datagram transmission.
3. An auxiliary task named Translate_ TOS_ Task, which operates in parallel with INM_ OUT, the main task, by translating type-of-service information from host-level to local-net level encoding.

The specifications for these three tasks are found in the specification part of Inm_ Out_ Module. The body parts of these three tasks are represented as stubs in the body part of Inm_ Out_ Module and the actual body parts of these tasks are listed in separate compilation units. (See Figure 4-3.)

```
                 Inm_Out_Module
 --------------------------------------------
|                                            |
|                Inm_Out                     |
|     ----------------------------------     |
|    |          The main task           |    |
|    |__________________________________|    |
|                                            |
|          Read_Init_Parameters              |
|     ----------------------------------     |
|    |          Auxiliary task          |    |
|    |__________________________________|    |
|                                            |
|          Translate_TOS_Task                |
|     ----------------------------------     |
|    |          Auxiliary task          |    |
|    |__________________________________|    |
|                                            |
|____________________________________________|
```

Figure 4-3: The three tasks embedded in Inm_ Out_ Module.

4.5. Important local procedures of Inm_ Out_ Module

Acitivity initiated within the main task (Inm_ Out) is delegated in two ways: (a) by entry calls to Read_ Init_ Parameters, and (b) by calls to one "principal" procedure defined in the body part of the containing package (Inm_ Out_ Module). This procedure is: Do-send, which in turn issues calls on other three others procedures, locally define (in Do_ send. These are. Read_ in_ header, Compact_ Options and Send_ fragment. The respective purpose of each of these principal and subsidiary procedures is spelled out in the commentary of their respective specification parts which are found in the specification part of Do_ send. The body parts of these procedures are represented as stubs in the body part of Do_ send and appear as separate compilations units in the listed code.

4.6. Section summary

This ends our short description, or "road map" through the code proper. There are 14 separate compilation units given in the Appendix. These are:

```
 1.  In_Out_Srv_Defs          -- Top-level definition package.
 2.  Inm_In_Out_Defs          -- Second-level definition package.
 3.  Inm_Out_Defs             -- Second-level definition package.

 4.  Memory_Module            -- Auxiliary module package.
 5.  Fifo_Module              -- Auxiliary module package.
 6.  Local_Net_Module         -- Auxiliary module package.

 7.  Inm_Out_Module           -- The main package.

 8.  Inm_Out                  -- The main task.

 9.  Read_Init_Parameters     -- Auxiliary task used by the
                                   main task, Inm_Out.
10.  Translate_TOS_Task       -- Auxiliary task used by the
                                   procedure Read_in_header.

11.  Do_send                  -- Procedure local to Inm_Out_Module
                                   called by Inm_Out.

12.  Read_in_header           -- Procedure local to Inm_Out_Module.
                                   called by Do_send.
13.  Compact_options          -- Procedure local to Inm_Out_Module
                                   called by Do_send.
14.  Send_fragment            -- Procedure local to Inm_Out_Module
                                   called by Do_send.
```

Appendix

```
------------------------------------------------------------------------
--                                                                    --
--                      Ada-to-Silicon Project                        --
--                       University of Utah:                          --
--                                                                    --
--              DoD Internet Protocol INM_OUT submodule               --
--                                                                    --
--        Ada code for the top-level definition package named:        --
--                        In_Out_Srv_Defs                              --
--                                                                    --
--                    Version of November 1, 1982                     --
------------------------------------------------------------------------
package In_Out_Srv_Defs is
   --
   -- Function:
   --    This package contains definitions needed by the INM_IN, INM_OUT, and
   --    INM_SRV modules.

   -- Useful bit-field types.

     subtype bit1  is     integer range 0..1;
     subtype bit3  is     integer range 0..7;
     subtype bit4  is     integer range 0..15;
     subtype bit8  is     integer range 0..255;
     subtype bit13 is     integer range 0..8191;
     subtype bit16 is     integer range 0..65535;
     subtype bit21 is     integer range 0..2097151;
     subtype bit24 is     integer range 0..16777215;
     subtype bit32 is     integer range 0..4294967295;

     shift1:              constant := 2;
     shift3:              constant := 8;
     shift4:              constant := 16;
     shift5:              constant := 32;
     shift6:              constant := 64;
     shift8:              constant := 256;
     shift13:             constant := 8192;
     shift16:             constant := 65536;

     subtype octet_type      is bit8;
     type octet_buffer_type is array(integer range <>) of octet_type;

-- The following code had been added to make the unchecked conversion routines
-- work.  Normally the default storage ( in the I432 ) for integers that are
-- less than or equal to 16 bits is a short ordinal (16 bit field).
-- So, normally converting a record of 2 bit8 integers to a bit16
-- integer would be equivalent to trying to stuff 2 short ordinals
-- into a single short ordinal.  The representation specifications fix this
-- problem.


     -- Representation specifications section.

     byte : constant integer := 0;

     for bit1'size  use 1;
     for bit3'size  use 3;
     for bit4'size  use 4;
     for bit8'size  use 1*byte;
     for bit13'size use 1*byte + 5;
     for bit16'size use 2*byte;
     for bit21'size use 2*byte + 5;
     for bit24'size use 3*byte;
     for bit32'size use 4*byte;

end In_Out_Srv_Defs;
```

```
----------------------------------------------------------------------
--
--
--                    Ada-to-Silicon Project                        --
--                    University of Utah:                           --
--                                                                  --
--           DoD Internet Protocol INM_OUT submodule                --
--   Ada code for the intermediate-level definition package named:  --
--                    Inm_In_Out_Defs                               --
--                                                                    --
--               Version of November 1, 1982                        --
----------------------------------------------------------------------  --
with In_Out_Srv_Defs;

use  In_Out_Srv_Defs;

package Inm_In_Out_Defs is
  --
  -- Function:
  --    This definition package contains definitions used by both
  --    the INM_OUT and INM_IN modules.


  -- INM data for communication with server


  max_header_length:          constant :=  64;
  max_segment_length:         constant :=  2 ** 16 - 20; -- Arbitrary.
  header_buffer_low_address:  constant :=  0;
  header_buffer_high_address: constant :=  max_header_length - 1;
  subtype header_ptr is integer range header_buffer_low_address ..
                                      header_buffer_high_address;
  subtype header_octet_buffer_type
                          is  octet_buffer_type(header_ptr);
  subtype header_length_type is  integer range 0 .. header_buffer_high_address
                                           - header_buffer_low_address + 1;
  type two_octet_record is
    record
      lo: octet_type;
      hi: octet_type;
    end record;

  type header_buffer_type is
    record
      version:             bit4;
      IHL:                 bit4;
      type_of_service:     bit8;
      total_length:        two_octet_record;
      identification:      two_octet_record;
      flags:               bit3;
      fragment_offset:     bit13;
      time_to_live:        bit8;
      protocol:            bit8;
      header_checksum:     two_octet_record;
      octet_buffer:        octet_buffer_type(12 .. header_buffer_high_address);
                                   -- The first eight of these octets
                                   -- consists of:
                                   --    source_address:      bit32;
  end record;                      --    destination_address: bit32;

-- INM data for communication with LNM

first_checksum_byte:         constant :=  10;
second_checksum_byte:        constant :=  11;
```

v o o

```
max_inm_packet:                   constant := 128; -- Octets (arbitrary).
                                                    -- ????????? E.I.D. 576???
subtype header_words  is  integer range  5 .. 16; -- Header length in words.
subtype header_octets is  integer range 20 .. 64; -- Header length in words.

-- Functions:
------------
function "xor"(
    first_operand:  octet_type;
    second_operand: octet_type)
  return octet_type;

function "xor"(
    operand1: two_octet_record;
    operand2: two_octet_record)
  return two_octet_record;


function Mask(
    number_to_be_masked_formal: integer;
    mask_formal:                integer)
  return integer;
  --
  -- Function:
  --   Performs a bit wise AND operation on
  --   the two passed parameters and returns the integer result.


function Shift_right(
    number_to_be_shifted: integer;
    shift_distance:       integer range 1 .. 15)
  return integer;
--
-- Function: Does equivalent of integer divide of number_to_be_shifted
-- by   2 ** shift_distance   returning the equivalent of the quotient
-- on unsigned (positive) integers.

-- Representation specifications section.

for two_octet_record use
  record
    lo at 0 range 0 .. 7;
    hi at 1 range 0 .. 7;
  end record;

end Inm_In_Out_Defs;

----------------------

package body In_Out_Defs is

  function "xor"(
      first_operand:  bit8;
      second_operand: bit8)
    return bit8
    --
    -- Function:
    --   Returns the Exclusive OR of two octets.
    --   The following implementation serves as a software guide only.
  is
    result, savea, saveb: bit8;
    abit, bbit:           bit8;
  begin
    savea  := first_operand;
    saveb  := second_operand;
    result := 0;
                                                -- Initialization.
    for index in 0 .. 7
    loop
```

```
      abit  := savea rem shift1;                     -- Get the least
                                                    -- significant bit.
      bbit  := saveb rem shift1;                     -- Get the least
                                                    -- significant bit.
      savea := Shift_right(savea, 1);                -- Strip off the least
                                                    -- significant bit.
      saveb := Shift_right(saveb, 1);                -- Strip off the least
                                                    -- significant bit.
      if not abit = bbit then                       -- Add the current xor bits
                                                    -- to the result.
        result := result + shift1 ** index;
      end if;
    end loop;
      return result;
  end;


  function "xor"(
      operand1: two_octet_record;
      operand1: two_octet_record)
    return two_octet_record
    --
    -- Function:
    --    Forms the exclusive OR for corresponding octets of two
    --    two_octet_operands.  Uses above declared "xor" function.
    --    I hope this is legal Ada. (Gary: Please check).  We use
    --    this function when performing checksumming on the full 16-bit
    --    checksums which are represented as two_octet_records.
  is
    resuld: two_octet_record;
  begin
    result.lo := operand1.lo  xor operand2.lo;
    result.hi := operand1.hi  xor operand2.hi;
    return result;
  end;


  function Mask(
      number_to_be_masked_formal: integer;
      mask_formal:                integer)
    return integer

    --   The following implementation serves as a software guide only.
  is
    first_number  : integer;
    second_number : integer;
    result        : integer;
    index         : integer;
    masking_done  : boolean;
  begin
    -- Initialize variables.

    first_number  := number_to_mask_formal;
    second_number := mask_formal;
    result        := 0;
    index         := 0;
    masking_done  := false;

    -- Do a bit by bit AND of both numbers starting from the
    -- low order bit.

    while not masking_done
    loop
      -- Test to see if both low order bits.

      if (first_number rem 2) = 1 and (second_number rem 2) = 1  then

        -- Add the current bit into the result.
        result := result + 2 ** index;
      end if;
      -- Take off the low order bit from both numbers.

      first_number  := Shift_right(first_number,  1);
      second_number := Shift_right(second_number, 2);

      -- If either number is zero then we are done.

      if (first_number = 0) or (second_number = 0)  then
        masking_done := true;
      else -- increment index
        index := index + 1;
      end if;
    end loop;
      return result;
  end mask;

  function Shift_right(
      number_to_be_shifted: integer;
      shift_distance:       integer range 1 .. 15)
    return integer
  --   The following implementation serves as a software guide only.
  is
  begin
    return number_to_be_shifted / shift_distance;
  end Shift_right;

end In_Out_Defs;
```

```
-----------------------------------------------------------------------------
--                                                                         --
--                       Ada-to-Silicon Project                            --
--                        University of Utah:                              --
--                                                                         --
--                 DoD Internet Protocol INM_OUT submodule                 --
--                                                                         --
--   Ada code for the intermediate-level definition package named:         --
--                          Inm_Out_Defs                                   --
--                                                                         --
--                     Version of November 1, 1982                         --
-----------------------------------------------------------------------------
with In_Out_Srv_Defs, Inm_In_Out_Defs;

use  In_Out_Srv_Defs, Inm_In_Out_Defs ;

package Inm_Out_Defs is
  --
  -- Function:
  --   This package contains definitions used in the INM_OUT module
  --   and the units to which it interfaces.


  --
  --
  --      Block Diagram of Anticipated Hardware Realization
  --
  --
  --      |               |            |               |
  --      |    INM_SRV    |<---------->|    MEMORY     |
  --      |               |            |               |
  --      -----------------            -----------------
  --        |  ^    |     ^              ^  |    ^    |
  --        |  |    |     |              |  |    |    |
  --       R| A|   X|    Y|              |  |    |    |
  --        |  |    |     |              |  |    |    |
  --        v  |    v     |              |  |    |    |
  --      ------------------      R      |  |    |    |
  --      |                |-------------   |    |    |
  --      |                |      A         |    |    |
  --      |                |<----------------    |    |
  --      |                |      X+1            |    |
  --      |                |---------------------     |
  --      |                |      B                   |
  --      |                |<--------------------------
  --      |    INM_OUT     |
  --      |                |      B
  --      |                |----------------------
  --      |                |      A              |
  --      |                |<-------------       |
  --      |                |      R       |      |
  --      |                |---------     |      |
  --      |                |        |     |      |
  --      ------------------        v     |      v
  --        |  ^    |     ^       ------------------
  --        |  |    |     |       |                |
  --        |  |    |     |       |      FIFO      |
  --       R| A|   S|    T|       |                |
  --        |  |    |     |       ------------------
  --        |  |    |     |          ^  |       |
  --        |  |    |     |         R| A|      B|
  --        v  |    v     |          |  v       v
  --      ---------------------------------------------
  --      |                                           |
  --      |               LN INTERFACE                |
  --      |                                           |



  -- Constants:
  max_tos_table_size: constant integer := 4    -- In octets.
```

```
                                        -- Actual size depends on
                                        -- available space in the
                                        -- hardware representation.
 max_local_net_tos_byte_size:  constant integer := 2;
                                     -- Max number of octets required
                                     -- to represent the local net TOS.
                                     -- We assume that 16 bits is more
                                     -- than sufficient to encode the
                                     -- local net tos.
                                     -- (Still not sure we need
                                     -- this constant. E.I.D.)
 early_ack:                  constant integer := 0;
 late_ack:                   constant integer := 1;

 segment_low_address:        constant :=  0;
 segment_high_address:       constant :=  max_segment_length - 1;

 -- Types used for intertask communication:
 ------------------------------------------
 x: constant integer := 4;   -- Data path widths: chunk of address.
                             -- SRV -> OUT and OUT -> MEMORY.

 -- Communication between the
 -- INM_OUT and MEMORY modules.

 subtype chunk_of_address_type is integer range 0 .. 2 ** x - 1;
                       -- Piece of start address for a datagram.
                       -- Each piece has x bits.

 type memory_request_type  is(
   load_address,
   receive_datum_octet);


 -- Communication between
 -- INM_SRV and INM_OUT modules.

 type srv_command  is(
   init_1,
   init_2,
   init_3,
   init_4,
   init_5,                   -- Not currently used.
   init_6                    -- Not currently used.
   init_7,                   -- Not currently used.
   send,
   test);

y: constant integer := 4; -- Data path width:
                            -- OUT -> SRV.
type out_response  is(
  sent_ok,
  dont_fragment_error,
  unsupported_tos,
  bad_header,
  bad_srv_command,
  local_net_time_out,
  local_net_error,
  other);

-- Communication among the
-- INM_OUT, LNM_OUT and FIFO modules.

s: constant := 4;    -- Data path width: OUT -> LN.

type local_net_command_type is(receive_fragment); -- Currently a set of one.


t: constant := 4;    -- Data path width: in -> out.
```

```
  type local_net_response_type is(
    fragment_received_ok,
    fragment_not_received);


  u: constant := 4;     -- Data path width: INM_OUT -> FIFO.

  type fifo_command_type is(
    reset,
    stnre,
    retrieve);

  -- Representation clauses.
  -------------------------
  for  memory_request_type   use(
    load_address           => 0,          -- Arbitrary choice. Hardware
    receive_datum_octet  => 1);           -- implementers may choose the reverse.

  for  srv_command   use(
    init_1  => 0,
    init_2  => 1,
    init_3  => 2,
    init_4  => 3,
    init_5  => 4,
    init_6  => 5,
    init_7  => 6,
    send    => 7,
    test    => 8);

  for  out_response   use(
    sent_ok                => 0,
    dont_fragment_error  => 1,
    unsupported_tos      => 2,
    bad_header           => 3,
    bad_srv_command      => 4,
    local_net_time_out   => 5,
    local_net_error      => 6,
    other                => 7);

  for  local_net_command_type  use(
    receive_fragment  => 0);

  for  local_net_response_type  use(
    fragment_received_ok   => 0,
    fragment_not_received  => 1);

  for  fifo_command_type  use(
    reset     => 0,
    store     => 1,
    retrieve => 2);

end Inm_Out_Defs;
```

```
---------------------------------------------------------------------------
--                                                                       --
--                        Ada-to-Silicon Project                         --
--                         University of Utah:                           --
--                                                                       --
--                 DoD Internet Protocol INM_OUT submodule               --
--                                                                       --
--               Ada code for the auxiliary package named:               --
--                             Memory_Module                             --
--                                                                       --
--                      Version of November 1, 1982                      --
---------------------------------------------------------------------------
with Inm_Out_Defs, In_Out_Srv_Defs;

use  Inm_Out_Defs, In_Out_Srv_Defs;

package Memory_Module is
  --
  -- Function:
  --   Represents the Memory module that holds to-be-sent datagrams
  --   as well as initialization parameters needed by INM_OUT.

  task Memory is
    --
    -- Function:
    --   Responds to Request entry call to either receive x-sized address
    --   bytes or send octets of information from the memory module to which
    --   it has access.  This task is a pure server, performing a memory
    --   function.

    entry Request(
        request_type_formal:         memory_request_type;
                                        -- Load_address or receive_datum_octet.
        chunk_of_address_formal:     chunk_of_address_type;
                                        -- Don't care when request_type_formal
                                        -- receive_datum_octet.
        octet_formal:            out octet_type);
                                        -- Don't care when load_address.
      --
      -- Function:
      --   When request_type_formal is receive_datum_octet, this entry copies
      --   an octet of information from a referenced location in its
      --   accessible memory, writes it into the octet_formal parameter,
      --   and then increments that reference.
      --   When request_type_formal is load_address, this entry
      --   "pursues construction" of a memory address by "taking in"
      --   the x-sized chunk of bits supplied by the first argument.
      --   The values input for the second or third parameters are
      --   "don't cares", when the first argument is, respectively,
      --   receive_datum_octet or load_address.
  end Memory;

end Memory_Module;
```

```
--------------------------------------------------------------------
--                                                                --
--                    Ada-to-Silicon Project                      --
--                     University of Utah:                        --
--                                                                --
--             DoD Internet Protocol INM_OUT submodule            --
--                                                                --
--                  Ada code for the package named:               --
--                          Fifo_Module                           --
--                                                                --
--                    Version of November 1, 1982                 --
--------------------------------------------------------------------
with In_Out_Srv_Defs, Inm_Out_Defs;

use  In_Out_Srv_Defs, Inm_Out_Defs;

package Fifo_Module is

  task Fifo is
    --
    -- Function:
    --   Server task only; issues no calls.

    entry Fifo_req(
        command_formal: fifo_command_type;
        octet_formal:   octet_type);

      --
      -- Function:
      --   This entry accepts the following command values:
      --   reset:    resets the FIFO
      --   store:    stores an octet in the FIFO
      --   retrieve: retrieves an octet from the FIFO
  end Fifo;

end Fifo_Module;
```

```
----------------------------------------------------------------------
--                                                                  --
--                      Ada-to-Silicon Project                      --
--                       University of Utah:                        --
--                                                                  --
--              DoD Internet Protocol INM_OUT submodule             --
--                                                                  --
--             Ada code for the auxiliary package named:            --
--                          Local_ Net_ Module                       --
--                                                                  --
--                    Version of November 1, 1982                   --
----------------------------------------------------------------------
with Inm_Out_Defs;

use  Inm_Out_Defs;

package Local_Net_Module is

  task Local_Net is
    --
    -- Function:
    --   This task represents the local net module, which can receive
    --   and return responses.

    entry Out_req(command_formal :        local_net_command_type;
                  response_formal:    out local_net_response_type);
      --
      -- Function:
      --   This entry recieves a value of command_formal from the Inm_Out task
      --   and passes back a result through response_formal.
      --   Command values are currently limited to only one value:
      --    receive_fragment.

  end Local_Net;

end Local_Net_Module;
```

```
-----------------------------------------------------------------------
--                                                                   --
--                      Ada-to-Silicon Project                       --
--                       University of Utah:                         --
--                                                                   --
--              DoD Internet Protocol INM_OUT submodule              --
--                                                                   --
--          Ada code for the main submodule package named:           --
--                          Inm_Out_Module                             --
--                                                                   --
--                    Version of November 1, 1982                    --
-----------------------------------------------------------------------

with Memory_Module,
     Inm_Out_Defs,
     Inm_In_Out_Defs,
     In_Out_Srv_Defs,
     Unchecked_conversion;


package Inm_Out_Module is
  --
  -- Function:
  --   This package contains task Inm_Out and an auxiliary procedure named
  --   Do_send. The task accepts commands from the SERVER module and acts
  --   to forward datagrams to the LOCAL NET module.

  use  Memory_Module, Inm_Out_Defs, Inm_In_Out_Defs, In_Out_Srv_Defs;

  -- Instances of Unchecked_conversion:
  ------------------------------------
  function Convert_twosome_array_to_record
                                        -- Used by Read_inheader.
  is
     new Unchecked_conversion(
            source  => octet_buffer_type(8 .. 1);
            target  => two_octet_record);

  function Convert_twosome_array_to_integer
                                        -- Used by Read_in_header.
  is
     new Unchecked_conversion(
            source  => octet_buffer_type(8 .. 1);
            target  => bit16);
                                        -- Used by Read_in_header.

  function Convert_two_octet_record_to_integer
                                        -- Used in Do_send.
  is
     new Unchecked_conversion(
            source  => two_octet_record;
            target  => bit16);

  function Convert_integer_to_two_octet_record
                                        -- Used in Do_send.

  is
     new Unchecked_conversion(
            source      => bit16,
            target      => two_octet_record);


  function Convert_srv_command_to_chunk_of_address
                                        -- Used by various.
  is
     new Unchecked_conversion(
            source  => srv_command;
            target  => chunk_of_address_type);
  -- Renamed task entry
  ---------------------
  procedure Memory_request(
      request_type_formal:           memory_request_type;
      chunk_of_address_formal:       chunk_of_address_type;
      octet_formal:              out octet_type)
    renames Memory.Request;

  -- Embedded task:  the "main show"
  ---------------------------------
  task Inm_Out is
    --
    --   This is the principal task of INM_OUT.
    --   It issues calls on the Go entry of Read_init_parameters and on
    --   Out_req entries in MEMORY, FIFO and Lmn_Out
    --   as well as Out_reset in FIFO.

    entry Srv_req(
        server_command_datum:      srv_command;
        response_to_server:    out out_response);
      --
      -- Function:
      --   This entry receives commands from INM_SRV module and
      --   passes back results through the parameter response_to_server.

  end Inm_Out;

  -- Embedded task:  an "auxiliary show"
  -----------------------------------
  task Read_init_parameters is

    entry Go(
        init_num_formal:      integer_range 0 .. 7;
        response:          out out_response);
      --
      -- Function:
      --   Gets init_num address chunks from INM_SRV and ships them over to
      --   the the associated Memory module, forming the base address of the
      --   storage block containing the initialization parameters; then
      --   gets the initialization parameters from the Memory module.
      --   Sets out_response to either send_ok if successful or to
      --   bad_srv_command if unsuccessful. (Can be unsuccessful if required
      --   tos table size exceeds available local space.)

    entry Srv_req(
        server_command_datum:      srv_command;
        response_to_server:    out out_response);
      --
      -- Function:
      --   This entry receives commands from the INM_SRV module.
      --   Note that task Inm_Out has an identical entry.

end Read_init_parameters;


  -- Embedded task:  another "auxiliary show"
  ----------------------------------------
  task Translate_TOS_Task is
    --
    -- Function:
    --   This pure server task executes concurrently with Inm_Out when
    --   performing a requested lookup in a globally accessible type_of_service
    --   translation table to determine, yes or nos, whether there is a
    --   local-net type-of-service corresponding to the given type-of-service.
    --   If yes, the matched local net tos value is indicated in the form of
    --   a returned index into the tos_table.  Send_fragment will then use
    --   this value later to flesh out the local net tos value to ship to the
    --   Fifo module.
  entry Begin_translation(
     inm_tos_byte:  bit8);
    --
    -- Function:
    --   This entry accepts the passed (from INM_SRV) TOS byte.
    --   The rendezvous is immediately broken to permit the calling task
    --   to resume computation.  In the "statement sequel" for this entry's
    --   accept statement, the server task performs the required lookup.
    --   For a successful completion of the search, the
    --   successful_translation flag is set to true, otherwise the flag
    --   is set to false.

  entry Send_result(
     successful_translation:  out boolean;
     tos_index:               out integer
                                    range 1 .. max_tos_table_size);

    --
    -- Function:
    --   Sends back the result of the immediately preceding Begin_translation
    --   entry call. If successful_translation is true, then tos_index
    --   references the tos_table element containing the corresponding
    --   local net tos value.

 end Translate_TOS_Task;


 -- Variable declarations:
 ------------------------
 last_result:                              out_response := sent_ok;

 time_out_in_milliseconds:                 integer range 1 .. 2**16 - 1;
                                              -- Computable from
                                              -- inm_time_out (see below)
                                              -- in procedure
                                              -- Read_init_parameters.
                                              -- Actually we may not
                                              -- compute it after all.

 local_net_tos_index: integer range 1 .. max_tos_table_size;
                                              -- Value received from call
                                              -- from Read_in_header on
                                              -- Translate_TOS_task.

   --Variables to hold initialization parameter values:

   inm_max_packet:                         two_octet_record;
                                              -- Largest size packet
                                              -- for the local net.
                                              -- Represented as a pair of
                                              -- octets and also used
                                              -- as a 16-bit integer after
                                              -- applying Unchecked_
                                              -- conversion.

   inm_address_length:                     octet_type;
                                              -- Used in Read_in_header.

   inm_time_out:                           two_octet_record;
                                              -- Waiting time at LN.
                                              -- Represented as a pair of
                                              -- octets and also used
                                              -- as a 16-bit integer after
                                              -- applying Unchecked_
                                              -- conversion.

   ack_type:                               octet_type;
                                              -- Early/late.

   local_net_type_of_service_table_row_size: octet_type;
```

```
    number_of_local_net_types_of_service:      octet_type;

    dont_care_octet:                           octet_type;
                          -- Used as an actual parameter for Memory.Request entry
                          -- calls when address chunks are being moved to the
                          -- memory module.
    -- Array:
    tos_table: octet_buffar_type(0 .. max_tos_table_size - 1);
                                                 -- The size of this tabla
                                                 -- depends on tha storage
                                                 -- space available in the

  -- Miscellaneous constants:
  ---------------------------
  dont_care_X_datum: constant chunk_of_address_type := 0;
                          -- Usad as an actual parameter for Memory.Request entry
                          -- calls when no addrass chunks are actually movad.
                          -- Hardware implementer may use indetarminate value.

end Inm_Out_Module;


package body Inm_Out_Module is

  procedure Do_send
    --
    -- Function:
    --    This procedure sends an internet datagram in the following steps:
    --    1) Reads the internet header.
    --    2) Translates internet TOS byte to a local net TOS.
    --    3) Constructs fragments and sends them to the local net.
    --         The option list for all but the first fragment are
    --         compacted, and the checksum for each fragment is computed.
    --
    --    Any encountered error terminates transmission of the datagram
    --    with an appropriate value assigned to the (global) variable, namad
    --    last_result.

  is separate;


  task body Inm_Out
  is separate;


  task body Translate_TOS_task
  is separate;


  task Read_init_parameters
  is separate;

end Inm_Out_Module;
```

```
-----------------------------------------------------------------------------
--                                                                         --
--                       Ada-to-Silicon Project                            --
--                        University of Utah:                              --
--                                                                         --
--               DoD Internet Protocol INM_OUT submodule                   --
--                                                                         --
--          Ada code for the body of the principal task named:             --
--                                                                         --
--                              Inm_Out                                    --
--                                                                         --
--                    Version of November 1, 1982                          --
-----------------------------------------------------------------------------
separate (Inm_Out_Module)

task body Inm_Out
  --
  -- Function:
  --   This is the principal task of INM_OUT.
  --   It issues calls on the Go entry of Read_Init_parameters and on
  --   Out_req entries in MEMORY, FIFO and Lmn_Out
  --   as well as Out_reset in FIFO.
is
  lcommand:         srv_command;
  init_num:         integer range 0 .. 7;
  dont_care_octet:  octet_type;                  -- Used as a dummy.
                                                 -- Hardware implementors
                                                 -- use an indeterminate
                                                 -- value.

begin

  -- Main command loop
  loop

    -- Get next command from the server.
    accept Srv_req(
        server_command_datum:     srv_command;
        response_to_server:    out out_response)
    do
      lcommand := server_command_datum;
      if  lcommand = test  then                -- Report last result.
        response_to_server := last_result;
      end if;
    end Srv_req;                               -- Break rendezvous.

    -- Now handle non-test srv_commands.
    case  lcommand  is
      when init_1 | init_2 | init_3 | init_4  =>
        case  lcommand  is
          when init_1 =>
            init_num := 1;
          when init_2 =>
            init_num := 2;
          when init_3 =>
            init_num := 3;
          when init_4 =>
            init_num := 4;
          when others =>
            null;
        end case;

      -- Start up task Read_init_parameters.
      Read_init_parameters.Go(
                              init_num_formal  => init_num,
                              response         => last_result);

      -- End of init command processing.  If unsuccessful, the response
      -- to the SRV module will be bad_srv_command.
      when send =>                          -- Get and put (move) all but last
                                            -- addr_chunk for the address
                                            -- of the datagram from the
                                            -- SRV to the Memory module.

        -- Note: In the following two loops we have a glitch in that we
        --       are matching a server_command_datum to
        --       type chunk_of_address_formal.  Looks like we need to
        --       apply Unchecked_conversion.  This problem also arises
        --       in earlier versions of this task.

        for index in 1 .. init_num - 1
        loop
          accept Srv_req(
              server_command_datum:      srv_command;
              response_to_server:    out out_response)
          do
            lcommand := server_command_datum;
            Memory_request(
                request_type_formal      => load_address,
                chunk_of_address_formal => server_command_datum,
                octet_formal             => dont_care_octet);
          end Srv_req;
        end loop;

        -- Last addr_chunk of datagram address is a special case, depending
        -- on ack_type in effect.

        accept Srv_req(
            server_command_datum:      srv_command;
            response_to_server:    out out_response)
        do
          Memory_request(
              request_type_formal      => load_address,
              chunk_of_address_formal => server_command_datum,
              octet_formal             => dont_care_octet);

          -- Late_ack case, where srv is held up till in consumes datagram.
          if ack_type = late_ack then
            Do_send;                          -- Do all remaining processing for
                                              -- sending this datagram.
          end if;
        end Srv_req;

        -- Now early_ack case, where srv is not held up.
        if ack_type = early_ack then
          Do_send;                            -- Do all remaining processing for
                                              -- sending this datagram.
        end if;

      when others =>
        last_result := bad_srv_command;

    end case;

  end loop;

end Inm_Out;                  -- end of task body
```

```
---------------------------------------------------------------------------
--                                                                       --
--                        Ada-to-Silicon Project                         --
--                         University of Utah:                           --
--                                                                       --
--                                                                       --
--              DoD Internet Protocol INM_OUT submodule                  --
--                                                                       --
--         Ada code for the body of the auxiliary task named:           --
--                                                                       --
--           Read_ Init_ Parameters        (used by Inm_Out)             --
--                                                                       --
--                 Version of November 1, 1982                           --
---------------------------------------------------------------------------
separate (Inm_Out_Module)

task body Read_Init_Parameters is

  -- Accessed globals:
  -------------------
  -- number_of_local_net_types_of_service:         octet_type
  -- local_net_type_of_service_table_row_size:     octet_type
  -- tos_table:                                    octet_buffer_type

  -- Renamed task entry:
  ---------------------
     -- The package Memory_Module containing the task Memory holds
     -- to-be-sent datagrams as well as initialization parameters
     -- needed by INM_OUT.

  procedure Memory_request(
      request_type_formal:          memory_request_type;
                                       -- Load_address or receive_datum_octet.
      chunk_of_address_formal:      chunk_of_address_type;
                                       -- Don't care when request_type_formal
                                       -- receive_datum_octet.
      octet_formal:            out octet_type)
                                       -- Don't care when load_address.
    renames Memory.Request;

  -- Local variable declaration:
  ------------------------------
  -- The following variable is commented out. It appeared only in the
  -- "high-level" used to read in the TOS table.  See below.
  --  number_of_tos_table_octets: integer range 2 .. max_tos_table_size - 1;
  octet_register:                octet_type;

begin
  loop
    accept Go(
        init_num_formal:           bit4;              -- For Carter's paper
                                                      -- only; otherwise bit3

        response:          out out_response)
      do
        response := sent_ok;                          -- Also means init_ok.

        -- Get from the server all of the addr_chunks needed to form the base
        -- address in memory that holds the initialization parameters and
        -- sends these chunks to the Memory module.
        for index in 1 .. init_num_formal
        loop
          accept Srv_req(                             -- Get next address
                                                      -- chunk from the
                                                      -- Server Module.

              server_command_datum:     srv_command;
              response_to_server:   out out_response)
          do
            Memory_request(                           -- Put chunk out to the
                                                      -- Memory module.
          request_type_formal        => load_address,
          chunk_of_address_formal  =>
                                   Convert_srv_command_to_chunk_of_address
                                       (server_command_datum),
          octet_formal               => dont_care_octet);
    end Srv_req;
  end loop;

  -- Get the 6 individual initialization parameters (contained in the
  -- next 8 octets received) from the Memory Module.
  for index in 1 .. 8
  loop

    Memory_request(
        request_type_formal      => receive_datum_octet,
        chunk_of_address_formal  => dont_care_X_datum,
        octet_formal             => octet_register);

    case index is
      when 1 => inm_max_packet.lo           := octet_register;
      when 2 => inm_max_packet.hi           := octet_register;
      when 3 => inm_address_length          := octet_register;
      when 4 => inm_time_out.lo             := octet_register;
      when 5 => inm_time_out.hi             := octet_register;
      when 6 => ack_type                    := octet_register;
      when 7 => local_net_type_of_service_table_row_size
                                            := octet_register;
      when 8 => number_of_local_net_types_of_service
                                            := octet_register;
    end case;
  end loop;

  -- Convert the local net timeout into milliseconds.?
  -- time_out_in_milliseconds := inm_time_out / 1000.0;
                                    -- Left-hand side variable declared
                                    -- in Inm_Out_Module. Value is used
                                    -- later in Do_send procedure.
                                    -- Note: Davis never did this in
                                    -- his design. Is this step needed?
                                    -- No! We don't need this step
                                    -- since the  quotient can be
                                    -- approximated by a div by 2**10
                                    -- in the event we need to
                                    -- represent milliseconds.

  -- Read in type of service translation table.

    --   The following code in comments is replaced below by a
    --   "lower-level" version that closely reflects the hardware
    --   implementation chosen in which we eliminate the need for
    --   for a multiplier.

--      number_of_tos_table_octets := local_net_type_of_service_table_row_size
--                                  * number_of_local_net_types_of_service;

--      -- Check to see if required table size exceeds maximum
--      if  number_of_tos_table_octets > max_tos_table_size  then
--        response := bad_srv_command;
--        return;
--      end if;

--      for index in 1 .. number_of_tos_table_octets
--      loop

--        Memory_request(
--            request_type_formal      => receive_datum_octet,
--            chunk_of_address_formal  => dont_care_X_datum,
--            octet_formal             => tos_table(index));
--      end loop;
```

```
        declare
          row_number: integer range 0 .. number_of_local_net_types_of service;
          col_number: integer range 0 ..
                                         local_net_type_of_service_table_row_size;
          index:      integer range 0
                                       .. number_of_local_net_types_of service
                                       * local_net_type_of_service_table_row_size
                                       := 0;
        begin
          row_number := 0;
          loop                       -- Outer loop reads all rows of TOS table.
            col_number := 0;
            loop                     -- Inner loop reads in one row of TOS table.
              Memory_request(
                  request_type_formal      => receive_datum_octet,
                  chunk_of_address_formal  => dont_care_X_datum,
                  octet_formal             => tos_table(index));

              col_number := col_number + 1;
              exit when col_number = local_net_type_of_service_table_row_size;

              index := index + 1;
              if  index > max_tos_table_size  then
                response := bad_srv_command;
                return;              -- Exit the current accept statement.
              end if;
            end loop;                -- End inner loop.

            row_number := row_number + 1;
            exit when row_number = number_of_local_net_types_of_service;
          end loop;                  -- End outer loop.
        end;                         -- End declare block.

      end Go;                        -- End of init processing.

    end loop;                        -- End of outer-most (inifinite)
                                     -- loop.
  end Read_Init_Parameters;
```

```
-----------------------------------------------------------------------
--                                                                   --
--                      Ada-to-Silicon Project                       --
--                        University of Utah:                        --
--                                                                   --
--               DoD Internet Protocol INM_OUT submodule             --
--                                                                   --
--          Ada code for the body of the auxiliary task named:       --
--                                                                   --
--             Translate_TOS_Task       (used by Read_in_header)     --
--                                                                   --
--                     Version of November 1, 1982                   --
-----------------------------------------------------------------------
separate(Inm_Out_Module)

task body Translate_TOS_Task

is

  -- Local variable declarations:
  --------------------------------
  Index:          integer range 8 .. max_tos_table_size - 1;
  local_tos_byte: bit8;
  success:        boolean;

begin
  loop
    accept Begin_translation(inm_tos_byte:  bit8)
    do
      local_tos_byte := inm_tos_byte;
    end Begin_translation;              -- Break rendezvous.

    -- Search for the INM_TOS byte in the TOS translation table.
    success := false;                   -- Initialize for search.
    index := 8;
    declare
      row_number: integer range 8 .. number_of_local_net_types_of_service -1
                               := 8;
                                  -- The value of
                                  -- number_of_local_net_types_of_
                                  -- service is dynamically defined
                                  -- in previous action of the
                                  -- Read_Init_Parameters task.
    begin
      while  row_number < number_of_local_net_types_of_service
      loop
        -- Test for the local_tos_byte in the TOS translation table.
        if  tos_table(index) = local_tos_byte  then
          index := index + 1;              -- Index now points at
                                          -- local net tos.
          success := true;
          exit;
        else
          index := index + local_net_type_of_service_table_row_size;
        end if;
        row_number := row_number + 1;
      end loop;
    end;                                  End of declare block.
    -- End of sequel for preceding accept statement.

    accept Send_result(
        successful_translation:    out boolean;
        tos_index:                 out integer
                                       range 1 .. max_tos_table_size)
                                  -- tos_index value is sent to the
                                  -- global named "local_new_tos_index"
                                  -- for use by Send_fragment.
    do
      successful_translation := success;
      tos_index              := index;
    end Send_result;

  end loop;

end Translate_TOS_Task;
```

```
------------------------------------------------------------------------
--                                                                    --
--                      Ada-to-Silicon Project                        --
--                       University of Utah:                          --
--                                                                    --
--              DoD Internet Protocol INM_OUT submodule               --
--                                                                    --
--                Ada code for the body of the procedure:             --
--                                                                    --
--                             Do_send                                --
--                                                                    --
--                    Version of November 1, 1982                     --
------------------------------------------------------------------------
with Inm_In_Out_Defs, Inm_Out_Defs;

use  Inm_In_Out_Defs, Inm_Out_Defs;

separate(Inm_Out_Module)


procedure Do_send is
  --
  -- Function:
    --   This procedure sends an internet datagram in the following steps:
    --   1) Gets the internet header from Memory_Module.
    --   2) Determines by entry calls to Translate_TOS_Task if the
    --      the  internet TOS byte corresponds to a valid local net TOS.
    --   3) Constructs fragments and sends them to the local net.
    --        The option list for all but the first fragment are
    --        compacted and the checksum for each fragment is computed.
    --
    --   Any encountered error terminates transmission of the datagram
    --   with an appropriate (explanatory) value assigned to the (global)
    --   variable, named last_result, declared in the Inm_Out_Module.

is
  -- Accessed globals:
  -------------------
  -- unsupported_tos:      out_response;
  -- bad_header:           out_response;
  -- dont_fragment_error:  out_response;


  -- Subtype declaration:
  ----------------------
  max_inm_address_size: constant := 2;    -- Size in octets.

  subtype inm_address_buffer_type is
                               octet_buffer_type(0..max_inm_address_size-1);

  -- Declarations of local variables:
  ----------------------------------
  inm_address_buffer:        inm_address_buffer_type;
  header_buffer:             header_buffer_type;
                                     -- Header record.

  header_octet_array:        header_octet_buffer_type;
                                     -- Octet array used to store header.
                                     -- In a hardware implementation, this
                                     -- array can be the same as the
                                     -- header_buffer.
  -- Need to insert here address clauses for both header_buffer and
  -- header_octet_array.

  header_length:             header_length_type;
                                     -- Header size in octets.
  segment_length:            integer range segment_low_address ..
                                               segment_high_address;
                                     -- Length of segment part of datagram
                                     -- in octets.
```

```
good_header_result:         boolean;    -- Result of the read_in_header call.
ok_tos_translation:         boolean;    -- Result of the tos_translation.

ok_fragment_transmission:   boolean;    -- Result of the Send_fragment call.
second_fragment:            boolean;    -- A flag that indicates if the
                                        -- current fragment is the second
                                        -- fragment of the current datagram.
more_fragments:             boolean;    -- A flag that indicates if there are
                                        -- more fragments to be formed.
fragment_length:            integer range 21 ..
                                  Convert_two_octet_record_to_integer
                                      (inm_max_packet);
                                        -- Used to indicate the current
                                        -- fragment's length.
current_fragment_offset:    integer range 0 .. 2 ** 16 - 1;
                                        -- Indicates the current fragment's
                                        -- offset into the overall data
                                        -- segment.
fragment_segment_length:    integer range 1 ..
                                  Convert_two_octet_record_to_integer
                                      (inm_max_packet) - 20;
                                        -- Used to indicate the length of the
                                        -- current fragment's data part.

datagram_total_length:      integer range 21 .. 2 ** 16 - 1;
                                        -- Used to save the total length of
                                        -- the current datagram.

checksum:                   two_octet_record;
checksum_with_options:      two_octet_record;
                                        -- Checksum values are developed
                                        -- in these auxiliary variables and
                                        -- later inserted into the
                                        -- header_buffer prior to copying
                                        -- the header to the Fifo module.

-- Constants:
------------
fragment_bit_true:          constant integer := 1;
                                        -- Used to set the more_fragments bit
                                        -- in header_buffer.flags.
do_not_fragment_true:       constant integer := 2;
                                        -- Used to test if the flags field
                                        -- indicates that no fragmentation
                                        -- is to occur.
-- Local procedures and functions:
---------------------------------
procedure Read_in_header(
    good_header: out boolean)
  --
  -- Function:
  --   This procedure first reads in the local net address of the
  --   the datagram into a local net address buffer and then reads in the
  --   datagram header octet by octet into a header buffer. Upon
  --   successfully completing the transfer of the header, the flag
  --   good_header is set to true; otherwise it is set to false.
is separate;


procedure Compact_options
  --
  -- Function:
  --   This procedure is invoked when constructing the second fragment.
  --   The procedure compacts the list of options in the header by keeping
  --   only those options that are flagged to be copied.  The header
  --   length and total length are also updated.
is separate;

procedure Send_fragment(
    data_fragment_size:                 bit16;
```

```
      successful_fragment_transmission: out boolean;
      explanation:                      out out_response)
    --
    -- Function:
    --   This procedure puts into the local net FIFO the following -
    --   1) local net address - local net address for the current fragment
    --   2) local net TOS     - local net TOS for the current fragment
    --   3) fragment header
    --   4) fragment data - which is pulled out byte by byte from the Memory
    --        associated with the INM_SRV module.  The size of
    --        the data fragment is passed as a parameter to this procedure.
    --
    --   This procedure, after stuffing the FIFO, will do a timed entry call
    --   on the local net (the call must be completed in the time specified
    --   by a parameter passed down from INM_SRV).  Upon successful
    --   transmission of the contents of the FIFO to the local net, the
    --   successful_fragment_transmission flag will be set to true; otherwise
    --   it is set to false. The value assigned to "explanaation" confirms
    --   the success (sent_ok) or provides the reason for failure.
  is separate;

  function Minimum(
      first_operand:   integer;
      second_operand:  integer)
    return integer
  is
    --
    -- Function:
    --   This function takes 2 operands and returns the minimum of the
    --   operands.
  begin
    if first_operand > second_operand then
      return second_operand;
    else
      return first_operand;
    end if;
  end Minimum;

------------ Body of Do_send begins here.  -------------------------------------
begin

  Read_in_header(good_header => good_header_result);

  if not good_header_result then
    last_result := bad_header;

    return;
  end if;

  if not (Convert_two_octet_record_to_integer(
            header_buffer.total_length)          >
         Convert_two_octet_record_to_integer(
            inm_max_packet) )    then

------------ Begin "single packet" case.

    -- Transfer checksum_with_options, whose value was computed
    -- by Read_in_header, into the proper slot in the header_buffer.
    header_buffer.header_checksum := checksum_with_options;

    Send_fragment(
        data_fragment_size               => segment_length,
        successful_fragment_transmission => ok_fragment_transmission,
        explanation                      => last_result);
    return;
  end if;

------------ End "single packet" case.
```

```
------------ Begin "multiple packet" (two or more fragments) case.

-- Fragment the datagram.
if  header_buffer.flags = do_not_fragment_true  then
  last_result := dont_fragment_error;
  return;
end if;

-- Initialize fragmentation variables.
current_fragment_offset   := 8;
second_fragment           := false;
more_fragments            := true;
ok_fragment_transmission  := true;
datagram_total_length     := Convert_two_octet_record_to_integer
                                 (header_buffer.total_length);

-- Back out octet containing old flags from the checksum.
checksum_with_options.lo := checksum_with_options.lo
                            xor header_octet_array(6);

-- Set more fragments flag in header_buffer.
header_buffer.flags      := fragment_bit_true;

-- Update checksum with octet containing new flags value.
checksum_with_options.lo := checksum_with_options.lo
                            xor header_octet_array(6);

while more_fragments and ok_fragment_transmission
loop
  if  second_fragment  then
    Compact_options;
    second_fragment := false;
  end if;

  fragment_length := Minimum(
                         first_operand  => datagram_total_length
                         second_operand =>
                                   Convert_two_octet_buffer_to_integer
                                        (inm_max_packet) );
  fragment_segment_length :=  fragment_length - header_length;

  -- Insert new total length into the header and update checksum.
  -- First back out octets containing total_length from the checksum.
  checksum_with_options   := checksum_with_options
                             xor header_buffer.total_length;

  header_buffer.total_length := Convert_integer_to_two_octet_record
                                   (fragment_length);

  -- Now update checksum with octets containg new total_length.
  checksum_with_options := checksum_with_options
                           xor header_buffer.total_length;

  -- Test to see if we are sending out the last fragment.
  if  current_fragment_offset + fragment_segment_length =
        segment_length  then
                                          -- If a < condition, then we
                                          -- then we still have another
                                          -- fragment to transfer.
                                          -- We should not get a > value
                                          -- because the last fragment is
                                          -- computed to contain the
                                          -- the remaining octets of the
                                          -- data segment.

    -- Clear more fragments bit and adjust checksum as well.

    -- First back out octet containing old flags from the checksum.
    checksum_with_options.lo := checksum_with_options.lo
                                xor header_octet_array(6);
```

```
      header_buffer.flags := 8;

      -- Now update checksum with octet containing new flags value.
      checksum_with_options.lo := checksum_with_options.lo
                                  xor header_octet_array(6);
    end if;


    -- Insert a new fragment offset into the header and also adjust checksum.

    -- First back out octets containing fragment offset from the checksum.
    checksum_with_options := checksum_with_options
                             xor Convert_twosome_array_to_record(
                                    header_octet_array(6 .. 7) );

    header_buffer.fragment_offset := current_fragment_offset;

    -- Now update checksum field in header_buffer with octets updated for
    -- new fragment offset.
    header_buffer.header_checksum := checksum_with_options
                                     xor Convert_twosome_array_to_record(
                                            header_octet_array(6 .. 7) );


    Send_fragment(
        data_fragment_size                 => segment_length,
        successful_fragment_transmission => ok_fragment_transmission,
        explanation                        => last_result);


    -- Set up parameters for the next time through the loop.
    if current_fragment_offset = 8 then
      second_fragment := true;
    end if;

    current_fragment_offset :=
                             fragment_segment_length
                             + current_fragment_offset;

    if not (current_fragment_offset < segment_length) then
      more_fragments := false;
    end if;

  end loop;
  -------------- End "multiple packet" case.

end Do_send;
```

```
-----------------------------------------------------------------------
--                                                                   --
--                     Ada-to-Silicon Project                        --
--                      University of Utah:                          --
--                                                                   --
--           DoD Internet Protocol INM_OUT submodule                 --
--                                                                   --
--           Ada code for the body of the procedure:                 --
--                                                                   --
--            Read_in_header  (called by Do_send)                    --
--                                                                   --
--              Version of November 1, 1982                          --
-----------------------------------------------------------------------
separate (Inm_Out_Module.Do_send)

procedure Read_in_header
    (good_header:  out boolean)
  --
  -- Function:
  --   This procedure first reads in the local net address of the
  --   the datagram into a local net address buffer and then reads in the
  --   datagram header octet by octet into a header buffer. Upon
  --   successfully completing the transfer of the header the flag
  --   good_header is set to true otherwise it is set to false.
  --   In the course of reading in the header, it makes a pair of entry calls
  --   translate_tos_task to obtain the local net type of service, if any
  --   and also computes the checksums (one without and one with the options
  --   component.  These checksums are "accumulated" in two-octet records
  --   declared (and cleared) in Do_send and named checksum and
  --   checksum_with_options, respectively.
is

  -- Constants:
  ------------
  minimum_header_length: constant integer :=  20;
  high_4_bits          : constant := 240; -- Upper 4-bit mask for an octet.
  high_3_bits          : constant := 224; -- Upper 3-bit mask for an octet.
  low_5_bits           : constant :=  31; -- Low   5-bit mask for an octet.
  low_4_bits           : constant :=  15; -- Low   4-bit mask for an octet.

  high_octet_byte      : constant :=   0; -- High byte of two_octet_buffer.
  low_octet_byte       : constant :=   1; -- Low  byte of two_octet_buffer.

  -- Accessed globals:
  -------------------
  -- checksum:              two_octet_record; -- Declared in Do_send;
  -- checksum_with_options: two_octet_record;
  -- local_net_tos_index:   integer range 1 .. max_tos_table_size;

  -- Local variable declarations:
  ------------------------------
  octet                  : octet_type;
  two_octets             : octet_buffer_type(0 .. 1);

  -- Renamed procedures and functions:
  -----------------------------------
  procedure Memory_request(
      request_type_formal:          memory_request_type;
      chunk_of_address_formal:      chunk_of_address_type;
      octet_formal:            out octet_type)
    renames Memory.Request;

  function Mask(
      number_to_be_masked_formal: integer;
      mask_formal:                integer) return integer
    renames Inm_In_Out_Defs.Mask;

  -- Local function definition:
  ----------------------------
  function Even(
      operand: integer)
    return boolean
  is
  begin
    if  operand rem 2  = 0  then
      return true;
    else
      return false;
    end if;
  end Even;


begin

  good_header := true;

  -- Get the local net address.  By convention, this field always precedes
  -- the actual datagram to be sent.
  for index in 0 .. inm_address_length - 1
  loop
    Memory_request(
        request_type_formal      => receive_datum_octet,
        chunk_of_address_formal  => dont_care_X_datum,
        octet_formal             => inm_address_buffer(index));
  end loop;

  -- Get the header's version number and length.
  Memory_request(
      request_type_formal      => receive_datum_octet,
      chunk_of_address_formal  => dont_care_X_datum,
      octet_formal             => octet);

  header_buffer.version := mask
                             (number_to_be_masked_formal  => octet,
                              mask_formal                 => low_4_bits);
  header_buffer.IHL     := mask
                             (number_to_be_masked_formal  => octet,
                              mask_formal                 => high_4_bits);

  -- Check the header version number.
  if  not (header_buffer.version =  4)  then
    good_header := false;
    return;

  elsif  header_buffer.IHL * 4  <  minimum_header_length  then
    good_header := false;
    return;
  end if;

  -- Update octets of the two checksums.
  checksum.lo              := octet xor checksum.lo;
  checksum_with_options.lo := octet xor checksum.lo;

  -- Get the type of service octet.
  Memory_request(
      request_type_formal      => receive_datum_octet,
      chunk_of_address_formal  => dont_care_X_datum,
      octet_formal             => header_buffer.type_of_service);

  -- We make the first entry call on translate_tos_task.
  Translate_TOS_Task.Begin_translation(header_buffer.type_of_service);


  -- Get the total length half word (2 octets).
  for index in 0..1
  loop
    Memory_request(
        request_type_formal      => receive_datum_octet,
        chunk_of_address_formal  => dont_care_X_datum,
        octet_formal             => two_octets(index));
```

```
end loop;

header_buffer.total_length := Convert_twosome_array_to_record
                                (two_octets(0..1));

-- Compute the segment's length in octets.
segment_length
  := Convert_twosome_array_to_integer(two_octets(0..1)) - header_length;

-- Update the two checksums.
checksum              := checksum  xor
                         Convert_twosome_array_to_record(two_octets(0..1));
checksum_with_options := checksum_with_options  xor
                         Convert_twosome_array_to_record(two_octets(0..1));

-- Get the identification half word (2 octets).
for index in 0..1
loop
  Memory_request(
      request_type_formal      => receive_datum_octet,
      chunk_of_address_formal  => dont_care_X_datum,
      octet_formal             => two_octets(index));
end loop;

header_buffer.identification := Convert_twosome_array_to_record(
                                  two_octets(0..1));
-- Update the two checksums.
checksum              := checksum  xor
                         Convert_twosome_array_to_record(two_octets(0..1));
checksum_with_options := checksum_with_options  xor
                         Convert_twosome_array_to_record(two_octets(0..1));

-- Get the flags (3 bits) and the fragment offset (13 bits).
for index in 0..1
loop
  Memory_request(
      request_type_formal      => receive_datum_octet,
      chunk_of_address_formal  => dont_care_X_datum,
      octet_formal             => two_octets(index));
end loop;

header_buffer.flags    := mask
                          (number_to_be_masked_formal =>
                               two_octets(high_octet_byte),
                           mask_formal          =>   high_3_bits);

header_buffer.fragment_offset :=
  mask(
       number_to_be_masked_formal  => two_octets(high_octet_byte),
       mask_formal                 => low_5_bits)
  * shift8 + two_octets(low_octet_byte);

-- Update the two checksums.
checksum              := checksum  xor
                         Convert_twosome_array_to_record(two_octets(0..1));
checksum_with_options := checksum_with_options  xor
                         Convert_twosome_array_to_record(two_octets(0..1));

-- Get the time-to-live octet.
Memory_request(
    request_type_formal      => receive_datum_octet,
    chunk_of_address_formal  => dont_care_X_datum,
    octet_formal             => header_buffer.time_to_live);

-- Update the two checksums.
checksum.lo              := checksum.lo  xor
                            header_buffer.time_to_live;
checksum_with_options.lo := checksum_with_options.lo xor
                            header_buffer.time_to_live;
-- Get the protocol octet.
```

```
Memory_requsst(
    request_type_formal        => receive_datum_octet,
    chunk_of_address_formal  => dont_care_X_datum,
    octet_formal               => header_buffer.protocol);

-- Update ths two checksums.
checksum.hi                := checksum.hi  xor
                              header_buffer.protocol;
checksum_with_options.hi := checksum_with_options.hi  xor
                              header_buffer.protocol;

-- Get the header checksum half word (2 octets) and dump it on the floor.
-- It's not needed.
for index in 0..1
loop
  Memory_request(
      request_type_formal        => receive_datum_octet,
      chunk_of_address_formal  => dont_care_X_datum,
      octet_formal               => two_octets(index));
end loop;

-- Get the source and destination addresses and the rest of the
-- header buffer which consists of the option octets.  For all octets
-- past the twentieth, update only one checksum. Note: no conversion
-- routine is needed here.


for index in 12 .. header_length - 1
loop
  Memory_request(
      request_type_formal        => receive_datum_octet,
      chunk_of_address_formal  => dont_care_X_datum,
      octet_formal               => header_buffer.octet_buffer(index));

  if     Even(index)   and then  index < 20 then
      checksum.lo                := checksum.lo  xor
                                    header_buffer.octet_buffer(index);
      checksum_with_options.lo := checksum_with_options.lo  xor
                                    header_buffer.octet_buffer(index);

  elsif  Even(index)   and then  index >= 20 then
      checksum_with_options.lo := checksum_with_options.lo  xor
                                    header_buffer.octet_buffer(index);

  elsif  not Even(index)  and then index  < 20 then
      checksum.hi                := checksum.hi  xor
                                    header_buffer.octet_buffer(index);
      checksum_with_options.hi := checksum_with_options.hi  xor
                                    header_buffer.octet_buffer(index);

  else  --  not Even(index) and then index >= 20 then
      checksum_with_options.hi := checksum_with_options.hi  xor
                                    header_buffer.octet_buffer(index);

  end if;

  -- We make the second entry call on Translate_TOS_Task.
  Translate_TOS_Task.Send_result(
      successful_translation => good_header,
      tos_index                => local_net_tos_index);
                                      -- Good_header is set false if
                                      -- translation is unsuccessful,
                                      -- in which case the value obtained
                                      -- for local_net_tos_index is
                                      -- will be ignored.

 end loop;

end Read_in_header;
```

```
-----------------------------------------------------------------------
--                                                                   --
--                      Ada-to-Silicon Project                       --
--                       University of Utah:                         --
--                                                                   --
--              DoD Internet Protocol INM_OUT submodule              --
--                                                                   --
--               Ada code for the body of the procedure:             --
--                                                                   --
--                 Compact_options      (called by Do_send)          --
--                                                                   --
--                    Version of November 1, 1982                    --
-----------------------------------------------------------------------
separate(Inm_Out_Module.Do_send)


procedure Compact_options
  --
  -- Function:
  --    This procedure is invoked when constructing the second fragment
  --    (and only the second fragment) of a datagram.
  --    The procedure compacts the list of options in the header by keeping
  --    only those options that are flagged to be copied.  The header length
  --    and total length are also updated as well as the checksum.  The
  --    value of checksum_with_options is recomputed from from the value of
  --    checksum
is
  -- Accessed globals.
  -------------------
  -- checksum:               two_octet_record;
  -- checksum_with_options:  two_octet_record;

  -- Subtype declaration:
  ----------------------
  subtype index6_type is integer range 0 .. 2 ** 6 - 1;
                                     -- Because max header size +  64 octets.

  -- Constants:
  ------------
  option_offset:                 constant integer := 20;
                                                   -- Offset (in octets)
                                                   -- indicating where the
                                                   -- options list begins.
  header_length_with_no_options: constant integer := 20;
  copy_option_true:              constant integer := 1;
                                                   -- Flag value indicating
                                                   -- that the current option is
                                                   -- be copied to all fragments.

  -- Local variable declarations:
  ------------------------------
  new_header_length:      index6_type;             -- In octets.
  options_length:         index6_type;             -- Length of options list.
  current_option_length:  index6_type;             -- Length of a candidate
                                                   -- option.
  leading_cursor:         index6_type;             -- Indicates next option
                                                   -- considered for copying.
  trailing_cursor:        index6_type;             -- Indicates slot in header
                                                   -- to receive the next
                                                   -- copied option.
  number_of_pad_octets: integer range 0 .. 3;

begin

  -- Does this header has any options?
  if header_length <= header_length_with_no_options then
    return;                                        -- There are no options to
  end if;                                          -- to "compact".

  -- Initialize variables.
  options_length  := header_length - header_length_with_no_options;
```

```
leading_cursor  := 0;
trailing_cursor := 0;

-- Initialize checksum_with_options from checksum.
checksum_with_options := checksum;

-- Begin compacting flagged options.

while leading_cursor < options_length          -- We use < rather than <=
                                               -- to avoid scanning the
                                               -- terminal octet, which must
                                               -- and "end-of-options-list"
                                               -- octet.
loop
  -- Is this option represented as a single or multiple octet?
  -- Discriminate by examining the option's number.
  if  header_buffer.octet_buffer(
        option_offset + leading_cursor)  rem  shift5   <  2  then
    current_option_length := 1;
  else

    -- Get the next option octet. It contains the option length as its
    -- value.
    current_option_length := header_buffer.octet_buffer(
                                 option_offset + 1 + leading_cursor);
  end if;

  -- Determine whether or not this option should be copied.
  if Shift_right(
        header_buffer.octet_buffer(option_offset + leading_cursor), 7)
      = copy_option_true  then
    for copy_index in 0  ..  current_option_length - 1
    loop
        header_buffer.octet_buffer(option_offset +
            trailing_cursor + copy_index)
          := header_buffer.octet_buffer(option_offset +
                leading_cursor + copy_index);
        -- Update checksum_with_options. Toggle on odd- and even-valued
        -- bytes in compacted options field.
        if  (trailing_cursor + copy_index) mod 2 = 0  then
          checksum_with_options.lo := checksum_with_options.lo xor
                                      header_buffer.octet_buffer(
                                          option_offset +
                                          trailing_cursor + copy_index);
        else
          checksum_with_options.hi := checksum_with_options.hi xor
                                      header_buffer.octet_buffer(
                                          option_offset +
                                          trailing_cursor + copy_index);
        end if;
    end loop;

    -- Update the trailing_cursor.
   trailing_cursor := trailing_cursor + current_option_length;
   end if;

   -- Update the leading_cursor.
   leading_cursor := leading_cursor + current_option_length;
end loop;

-- Pad out the last option word with pad octets (including the last one,
-- which is an end-of-all-options octet) until we have reached a 32-bit
-- boundary.

number_of_pad_octets := 4 - (trailing_cursor mod 4);
for copy_index in 0 .. number_of_pad_octets - 2
loop
-- Insert a "pad" octet  (= "00000001").
header_buffer.octet_buffer(option_offset + trailing_cursor + copy_index)
  := 1;
  -- Update checksum_with_options with pad octet (= "00000001").
  checksum_with_options := checksum_with_options  xor 1;
  end loop;

  -- Now insert the last pad octet.
    -- Insert an "end-of-all-options" octet  (= "00000000").
    -- Note that the zero value of the end-of-all-options octet
    -- will not change the value of the current checksum; hence there is
    -- no update of the checksum for this octet.
  header_buffer.octet_buffer(
      option_offset + trailing_cursor + number_of_pad_octets - 1)
    := 0;

  new_header_length
    := option_offset + trailing_cursor + number_of_pad_octets;

  -- Update the total length field and the checksum.

  -- First back out octets containing total_length from the checksum.
  checksum_with_options := checksum_with_options  xor
                           Convert_twosome_array_to_record(
                               header_octet_array(2 .. 3));

  header_buffer.total_length := Convert_integer_to_two_octet_record(
                                    Convert_two_octet_record_to_integer(
                                        header_buffer.total_length)
                                        - header_length - new_header_length);

  -- Now update checksum with octets containing new total_length.
  checksum_with_options := checksum_with_options  xor
                           Convert_twosome_array_to_record(
                               header_octet_array(2 .. 3));

  -- Update the IHL field and the checksum.
  -- Back out octet containing old IHL value from the checksum.
  checksum_with_options.lo := checksum_with_options.lo
                              xor header_octet_array(0);

  header_buffer.IHL := Shift_right(new_header_length, 4);

  -- Update checksum with octet containing new IHL value.
  checksum_with_options.lo := checksum_with_options.lo
                              xor header_octet_array(0);
end Compact_options;
```

```
-----------------------------------------------------------------------
--                                                                   --
--                       Ada-to-Silicon Project                      --
--                        University of Utah:                        --
--                                                                   --
--              DoD Internet Protocol INM_OUT submodule              --
--                                                                   --
--               Ada code for the body of the procedure:             --
--                                                                   --
--                   Send_fragment      (called by Do_send)          --
--                                                                   --
--                    Version of November 1, 1982                    --
-----------------------------------------------------------------------
with Fifo_Module, Local_Net_Module;

separate(Inm_Out_Module.Do_send)


procedure Send_fragment(
    data_fragment_size:                          bit16;
    successful_fragment_transmission:     out boolean;
    explanation:                          out out_response)
  --
  -- Function:
  --   This procedure puts into the local net FIFO the following -
  --   1) local net address - local net address for the current fragment
  --   2) local net TOS     - local net TOS for the current fragment
  --   3) fragment header
  --   4) fragment data - which is pulled out byte by byte from the Memory
  --        associated with the INM_SRV module.  the size of
  --        the data fragment is passed as a parameter to this procedure.
  --
  --   This procedure, after stuffing the FIFO, will do a timed entry call
  --   on the local net (the call must be completed in the time specified
  --   by a parameter passed down from INM_SRV).  Upon successful
  --   transmission of the contents of the FIFO to the local net, the
  --   successful_fragment_transmission flag will be set to true; otherwise
  --   it is set to false. The value assigned to "explanation" confirms
  --   the success (sent_ok) or provides the reason for failure.


 is
  -- Renamed task entries:
  -----------------------
  procedure Memory_request(
      request_type_formal:          memory_request_type;
      chunk_of_address_formal:      chunk_of_address_type;
      octet_formal:             out octet_type)
    renames Memory.Request;

  procedure Local_net_out_req(
      command_formal:      local_net_command_type;
      response_formal:  out local_net_response_type)
    renames Local_Net_Module.Local_Net.Out_req;

  procedure Fifo_req(
      command_formal:  fifo_command_type;
      octet_formal:    octet_type)
    renames Fifo_Module.Fifo.Fifo_req;


  -- Local variable declarations:
  -------------------------------
  octet_register:            octet_type;
  local_net_response:        local_net_response_type;

 begin
  successful_fragment_transmission := true;

  -- Reinitialize the FIFO.
```

```
Fifo_req(
    command_formal  => reset,
    octet_formal    => dont_care_octet);


-- Load the FIFO with the fragment's local net address previously
-- saved in the inm_address_buffer.
for index in 0 .. inm_address_length - 1
loop
  Fifo_req(
      command_formal  => store,
      octet_formal    => inm_address_buffer(index));
end loop;

-- Load the FIFO with the local net tos.
for index in local_net_tos_index ..
                              local_net_type_of_service_table_row_size - 1
loop
  Fifo_req(
      command_formal   =>  store,
      octet_formal     =>  tos_table(index));
end loop;

-- Load the fragment's header into the FIFO.
for index in 0 .. header_length - 1
loop
  Fifo_req(
      command_formal  => store,
      octet_formal    => header_octet_array(index));
end loop;

-- Get the data fragment from the memory and load it into the FIFO.
for data_index in 0 .. segment_length - 1
loop
  Memory_request(
      request_type_formal      => receive_datum_octet,
      chunk_of_address_formal  => dont_care_X_datum,
      octet_formal             => octet_register);

  Fifo_req(
      command_formal  => store,
      octet_formal    => octet_register);
end loop;



-- Do a timed entry call on the local net indicating that
-- the FIFO has a fragment with local net information in it.
select                                          -- Conditional select.
  Local_net_out_req(                             -- Was fragment received?
      command_formal  => receive_fragment,
      response_formal => local_net_response);
or
  delay time_out_in_milliseconds;                -- Value was computed by
                                                 -- Read_init_parameters

  -- The local net rendezous has timed out.
  explanation := local_net_time_out;
  successful_fragment_transmission := false;

end select;

-- Test to see if the local net received the fragment.
if  successful_fragment_transmission     -- Local net did not time out.
        and  then                          -- Ada "Short-circuit" phrase.
     not (local_net_response = fragment_received_ok)  then
                                           -- Local found something was wrong.
  explanation := local_net_error;
  successful_fragment_transmission := false;
end if;
end Send_fragment;
```

References

[1] Alan B. Hayes.
High-level Logic Design of the DoD INM-OUT Module.
April, 1982.
Ada to Silicon Project Internal Working Document, Department of Computer Science, University of Utah.

[2] Organick, E. I., and Lindstrom, G.
Mapping high-order language units into VLSI structures.
In *Proc. COMPCON 82*, pages 15-18. IEEE, Feb., 1982.

[3] Organick, E.I., Carter, T.M., Lindstrom, G., Smith, K.F., Subrahmanyam, P.A.
Transformation of Ada Programs into Silicon. SemiAnnual Technical Report.
Technical Report UTEC-82-020, University of Utah, March, 1982.

[4] Organick, E.I., Carter, T., Hayes, A.B., Lindstrom, G., Nelson, B.E., Smith, K.F., Subrahmanyam, P.A.
Transformation of Ada Programs into Silicon. Scond SemiAnnual Technical Report.
Technical Report UTEC-82-103, University of Utah, November, 1982.

[5] Postel, Jon: editor.
Internet Protocol: DARPA Internet Program, Protocol Specification.
Technical Report RFC 791, Information Sciences Institute, USC, Sept., 1981.

[6] Postel, Jon: editor.
Transmission Control Protocol: DARPA Internet Program, Protocol Specification.
Technical Report RFC 793, Information Sciences Institute, USC, Sept., 1981.

[7] Postel, Jon: editor.
Assigned Numbers.
Technical Report RFC 790, Information Sciences Institute, USC, Sept., 1981.

[8] Postel, Jon: editor.
Internet Control Message Protocol: DARPA Internet Program, Protocol Specification.
Technical Report RFC 792, Information Sciences Institute, USC, Sept., 1981.